CMMI Supports Enterprise-Wide Process Improvement

Lt. Col. Joe Jarzombek
ESIP Director

The success of a project is often dependent on the organizational processes and capabilities that cut across multiple disciplines. Several organizations already use one or more Capability Maturity Models (CMMs) to guide their process improvement efforts. However, process-related CMMs have lacked integration among disciplines, and process assessments have been known to result in inconsistent findings. Improvement efforts based on unique CMMs have resulted in suboptimization, confusion, and potentially unnecessary expenditure of process improvement resources [1]. These are some of the driving needs for the current collaborative effort to integrate CMMs (dubbed CMMI) that is sponsored by the Office of the Secretary of Defense for Acquisition and Technology, with participation by other government organizations, the Software Engineering Institute, and industry.

The CMMI team will produce a set of integrated products to support process and product improvement. It will include a framework to generate CMMI products, individual capability models, training and assessment products, as well as a tailoring guide, and a glossary to reflect common terminology among disciplines [2]. The CMMI team will develop a framework that will generate both staged and continuous capability models as explicitly specified in the CMMI "A" specification. The CMMI product suite is intended to preserve government and industry process improvement investments, enhance use and generation of multiple models, and accommodate considerations for consistency with Department of Defense directives and industry (Electronic Industries Alliance) standards as well as support international standards.

By integrating process areas among disciplines, the CMMI will better support the institutionalization of enterprise-wide process improvement, cutting across disciplines that are often compartmentalized within organizations. The Federal Aviation Administration has already demonstrated the value of using continuous representation of an integrated CMM with staging guidelines (http://www.faa.gov/ait/sepg). The use of continuous representation with staging guidelines leaves it to organizations to decide priority and ordering of processes to improve based on business objectives. It facilitates, through staging, a summarization of organizational maturity level based on experience with successful process improvement priorities.

Industry is a major proponent and participant in this CMMI effort, and industry involvement is increasing in the number of CMMI stakeholder reviewers. There are opportunities for participation in CMMI pilot projects planned to start in January 1999 that offer benefits to organizations. Those interested in learning up-to-date information about the CMMI, reviewing CMMI frequently asked questions, or who desire to participate in CMMI pilot projects should go to the SEI Web site, http://www.sei.cmu.edu, and select the CMM section. ◆

Capability Maturity Model and CMM are registered trademarks of Carnegie Mellon University.

References

1. Schaeffer, Mark D., "Capability Maturity Model Process Improvement," *CROSSTALK*, Software Technology Support Center, Hill Air Force Base, Utah, May 1998.
2. Schaeffer, Mark D., Philip Babel, Jack Ferguson, et al., "Overview of the Integrated Capability Maturity Model (CMMI) Development Project," panel presentation, Tenth Annual Software Technology Conference, Salt Lake City, Utah, April 22, 1998.

Apache Lessons in Acquisition Management

I read with interest "Slaying the Software Dragon," (*CROSSTALK*, May 1998) especially the figures that indicated an underwhelming response to the course offerings. I offer a slightly different viewpoint and analogy: Approximately 100 years ago, the U.S. Cavalry found itself unable to effectively deal with the Apache. The Army's tactical leaders (managers), having studied traditional war fighting (management), were having trouble understanding a "different" enemy (software development). The most effective response was not to try to make each of its tactical leaders (acquisition managers) into an effective tracker, but to hire the enemy: Apache Indian scouts. Today, we call these *independent validation and verification* agents (software projects) or *validation, verification, and accreditation* agents (software models)—experienced developers hired to search through the horse dung (documents) left by the enemy (developer) to try to fathom the enemy's intent (look for the issues, weaknesses, problems, etc., behind the smoke, mirrors, hand waving, and slideware).

Perhaps we should look at ways to use these individuals more effectively.

Joe Saur
Fort Monroe, Va.

Rules a Program Manager Can Live By
Getting Back to the Basics

Col. Wayne M. Johnson
U.S. Air Force

Program managers who focus on customers, money, and common sense are more likely to succeed than those who do not. This article offers 10 basic steps toward better program management that emphasize those three points of focus.

With acquisition reform, reengineering, downsizing, rightsizing, and old-fashioned turmoil, a lot of interest from public and private sectors now focuses on how we, as program managers, do business. Because of this increased focus and interest, many organizations are attempting to quantify what they do.

In response to this growing interest, my experience and that of many other program managers tells me we need to get back to the basics:

- What are we doing?
- How are we doing?
- How do we know?

Instituting planning charts, financial summary charts, and color-coded risk and status charts are not by themselves marks of progress. The team must understand and use the material. To do that, we need to employ the basics—planning a solid program using common sense and sound management techniques.

Some organizations appear to have lost sight of that. They are doing solid planning from a technical perspective but not applying those same disciplined techniques to the business side. The tools we use to plan, organize, and evaluate should be just that—tools ... but not an end in themselves. The rules that follow are not new, exciting, or particularly insightful, but they work, and may help you avoid some of the problems and pitfalls in getting the job done.

This article is based on an article that first appeared in Program Manager, November/December 1997.

Step 1 – Do Not Try to Impress People by Building a Better Mousetrap

In trying to express this concept in ways that are new and different, words fail me. Tired old phrases come to mind such as "Don't reinvent the wheel," and "Don't fix what isn't broken"—neither of which is likely to hold anyone's attention. Regrettably, there is no fascinating way to say what we all know to be true: The institutional resistance (inertia) of "not invented here" needs to be addressed upfront. Think about it. Why not borrow a good idea from another office, give the originator credit, improve the process or idea, and move on? This is much more efficient and productive than trying to come up with that one "brilliant idea" yourself.

With this philosophy in mind, keep an eye out for good ideas in your organization. As the old saying goes, "You'd be surprised what you can accomplish if you don't mind who gets the credit." So see what works, and keep it. If what you are doing or what you have tried is not working, start with Step 2.

Step 2 – Know Who the Customer Really Is

The customer is the one who is putting up the financial resources, right? Most of the time. As an example, the customer for U.S. Air Force fighters and bombers is the Air Combat Command (ACC). But when it comes to developing requirements, the acquisition community is ACC's customer. If the war fighter's requirements are not nailed down, how can you acquire a system that they will be happy with? Moreover, if the requirements constantly change and customers do not seem to know what they want, we have "requirements creep." And depending on who is in the meeting, requirements creep may be a noun or a verb.

Once your customer has understandable and definitive requirements, you must know what the cost, schedule, and performance parameters are and baseline the program. Without it, you will not be able to communicate what is required to successfully fulfill the requirement, and others will not know what to expect in return. Put another way, a baseline serves as the vehicle to establish and track a common set of expectations. When developing a program baseline that incorporates cost, schedule, and performance, do not forget that your project integrates with many other products and processes such as training, spare parts, or maintenance equipment.

Some people develop a baseline as a document. I like to think of it as a set of briefing charts (which helps me stifle my verbosity). Perhaps you will find, as I did, that most of the benefit of a simple baseline document (I recommend keeping it to six to eight charts) is in building it and coordinating with all the affected agencies, including the customer. As a forcing function, the baseline applies discipline in bringing

the program together and ensures that its strategy is supportable.

Step 3 – Get a Second Opinion

Suppose you went to the doctor for a standard check-up and received a dire prognosis. You would likely seek a second opinion. Likewise, if your program receives word that it has a sudden illness, but it seems fine, get a second opinion. I know of a program manager who went ballistic after hearing that a piece of government-furnished equipment (GFE) was not going to be available for his program. After being yelled at by the customer, he finally had to elevate it to his supervisor. The supervisor then called a different "expert" point of contact and found there was plenty of the GFE item available.

The lesson to be learned from this true situation again takes us back to the basics. When things unexpectedly look bad, get a second opinion. The same is true for those times when you believe things are headed the wrong direction and your single point of contact says, "Don't worry, be happy"—get that second opinion.

Step 4 – Realize All Software Development Is Moderate Risk

"What you see is not always what you get" is a general rule of software development. Although debugging and testing a program may reveal many hidden problems, these actions alone cannot guarantee that all problems are detected. Historically, software has proven itself difficult to scope and insidiously susceptible to requirements growth. Keep this axiom in mind: "The more complex your solution, the more vulnerable it is to simple problems." Be wary of magicians who claim that previously discovered hardware problems can be fixed with a simple software modification. One senior program director once told me his rule of thumb: No matter where you are in software development, you are always two years behind schedule and need twice as much money. Expect it, plan for it, and manage it.

Step 5 – Know Your Program's Status

Many organizations use color codes to communicate the status of the project. I have always been fascinated by the variety of definitions and the finite detail program managers use and confuse to define whether a program, project, or functional area is "green," "yellow," or "red." Depending on the management philosophy of the organization, green, yellow, or red is usually the program manager's assessment.

For example, problem N may be coded different colors by different program managers. If the program is one month behind, do you evaluate it red, yellow, or green? The color depends on when the customer needs it. Following are three simple definitions to consider when preparing color-coded future assessments.

- If the program or project fills your day, keeps you challenged, and is a reason why they need you in government service, the program is green.
- If you ponder the day's events on the drive home and know that your boss will be irritated to hear from someone other than you about the latest "fun" you are having, the program is yellow.
- If you find yourself waking up in a cold sweat in the middle of the night considering other employment options, hoping that your boss can help you fix all the problems, the program is red.

Step 6 – Follow the Money

Everything you do is connected to money, and if you did not control the funding no one would pay any attention to you. Start thinking of financial planning documents as program management planning documents because that is what they are. Always be familiar with your financial situation. For example, I have seen unintentional problems arise when several functional areas believed they were entitled to the same chunk of money but did not talk to each other about who really owned it. If you depend on too many good things to happen to be successful, you probably will not be. If you are not managing the money, you are not managing the program. That is always the bottom line.

Step 7 – Summarize Meetings

Have you ever sat through a one-hour meeting listening to all the attendees speak their mind? At the end of the meeting, with 15 suggestions from six people, it is difficult to know who plans to do what unless the program manager summarizes for the group what the course of action will be. If, at the end of your meeting, you have not summarized a plan of action, you might find yourself rescheduling another meeting. Get into the habit of summarizing each meeting and save time, effort, and a lot of headaches down the road.

Step 8 – Use the "Aunt Agnes" Test

A situation develops that requires you, the program manager, to make a decision. But does the course of action you are about to select make sense? In acquisition, we have surrounded ourselves with processes, integrated acronym lists (IAL), and program management review teams, all of which can deprive us of our common sense.

I have been taught to use this simple test: Pretend you have an Aunt Agnes who owns a farm in Iowa, where she grows corn. Can you explain the program and your decision to her? Would she understand it? Does it make sense? Can you defend the course of action to her? If the answer to any of these questions is "no," rethink your strategy because you are about to lose your way. And do not bother look up IAL—I made that up to show how unnecessary complexity will only confuse Aunt Agnes and your customer.

Step 9 – Make a Decision

We have all sat through meetings where a detailed, insightful discussion about the pros and cons of a project occurred to the nth degree. But in the end, no one knew what course of action to which the program manager

agreed. What did he want? Did she say, go ahead? The difference between the program manager and a lot of process-oriented staff help is that you are required to make decisions. Do not forget that. If you do, you will be without a job.

Sometimes no decision is the worst decision. Be careful not to get caught in this type of organizational paralysis. One senior leader once advised that "you need to go into the job assuming you have already been fired—only then will you be willing to make the right decisions." Take in the important details, look at the alternatives, understand the options, then make a decision and move on.

Step 10 – Manage, Do Not Micromanage

Stay focused on the goals and ideas that are important to you, and stick to the basics. Watch the details without micromanaging your team. You cannot always be there to answer the questions, so make sure your team knows what is going on. Treat everyone with respect. And have fun.

Being a program manager is a lot like being a utility infielder in baseball. You know what will make your effort successful, and you have a team of functional experts to help you along the way. Let them know what you expect from them, and chances are they will not let you down. Remember, these jobs are 10 percent expertise and 90 percent common sense. To win the game, stick to the basics, focus on your goal, and rely on teamwork.

Keep It Simple

You do not get paid more for making it complicated, so stick to the basics. The tools for becoming a more effective program manager that I have outlined in this article are quite simple. Every one of us has thought of them, but the working process can still be confusing. When you think you are losing control of a project, check to see if you are following these tips. Chances are you will quickly recognize how to fix it. ◆

About the Author

Col. Wayne M. Johnson was formerly chief of F-16 Programs for Turkey, Aeronautical Systems Center (ASC), Wright-Patterson Air Force Base, Ohio, where he managed the 240 aircraft, Foreign Military Sales Turkish F-16 weapons system. A command pilot with 2,800 hours of flying time, Johnson was the 1995 winner of the Air Force Association/ASC Sylvester Award for Program Management. In 1996, he graduated from the Advanced Program Management Course, Defense Systems Management College. He is currently program director at the Joint Airborne Signals Intelligence Program Office.

Voice: 937-255-9968 DSN 785-9968

The Softer Side of Project Management

Janice Strauss
National Security Agency

Many project managers limit themselves to techniques they have acquired through formal channels, which decreases their chances for success. I contend that there are many "softer" techniques available that have a great impact on a project. In this article, I share some of the techniques I use to increase the likelihood of achieving project goals.

Typically found in the toolbox of project management are techniques for cost estimation, risk management, meeting staff requirements, and establishing work breakdown structures. These techniques represent essential project management skills usually acquired through formal courses, reading, or on-the-job training. These learning methods often overlook the "softer side" of project management. Understanding this side constitutes yet another tool just as critical to project success as more formal ones. A manager's ability to effectively maintain morale, motivate the team, and use resources determines whether team members have a sense of pride in their project and feel ownership of it.

This article highlights some techniques I have used to address the human side of project management. Some focus on ensuring everyone on the team feels comfortable with their role. Others establish and maintain good team morale. All help a project maintain momentum toward a successful conclusion.

Soft Project Management Techniques

A new project is about to commence. The team consists of senior engineers and computer scientists, all with many years of experience in the tools that will be used on the project. This team also has a history of working together and keeping one another well informed. "A dream team," you think to yourself, and with good reason. Such a team is not likely to be found in the real world. It is much more probable that a project will have a blend of junior and senior employees with varying experience levels. Furthermore, the team will probably have little history with one another and with the technologies, thus requiring much groundwork to initiate the project.

Pair Team Members

Getting junior employees comfortable, up to speed, and productive quickly is definitely a challenge. Formal training helps, but this requires time and money that may not be available. In this situation, I pair junior, inexperienced team members with those who have more expertise. Junior persons may shadow their mentors, observing and studying their behavior, or the pairs may work on a task together, with the senior person handling the more difficult aspects and serving as a mentor to the junior person.

This technique pays for itself in the long run. On one project, a new developer initially played the junior role for a few months. When another inexperienced person joined the team, the first was able to move up to the senior role and successfully served as mentor. This transition was a source of great pride to the entire team.

Ensure Expert Technical Support

Dealing with today's world of constantly changing technologies can make any reputable manager cringe. No sooner has one committed to a suite of tools than a new and better solution becomes apparent. In the case of technologies like Java, new releases occur at short intervals—a daunting prospect for developers. On one of my projects, the team chose Java for its many advantages including hardware independence and enhanced programmer productivity, yet no one on the team had previously used this language. To manage the risk involved, I took steps to ensure that expert technical support for Java was available and accessible to the team. This came in two forms: First, I hired a Java mentor who provided guidance to the rest of the team, introduced new Java tools, and reviewed all Java software. Second, team members were also encouraged to maintain a close relationship with the vendor to stay aware of the latest developments and to provide them with requirements for new features. With this strategy in place, Java increased the team's productivity rather than proving to be an obstacle.

Assign People with Care

Have you ever felt that management views developers as interchangeable game pieces they can arbitrarily move between projects? Many times, I have seen people placed in critical positions based on their job title rather than their skills. Putting team members in positions they cannot handle usually leads to negative consequences in terms of schedule, quality, and productivity. Just because a person is hired as senior computer scientist does not mean that person can take on every task successfully and with little monitoring. Admittedly, there will be times when it is necessary to assign team members to tasks for which they do not have the right expertise. I do this with caution—only with people who have proven track records and in whom I have great confidence. I do not expect those with newly acquired skills to take on critical or complex tasks.

Consider work habits when assigning tasks. Some people work faster than others, thrive on challenge, and withstand pressure well. Others proceed at a more cautious pace and prefer to work on the familiar. Take all these factors into account to prevent situations in

which employees are frustrated with their assignments and cannot make a contribution.

Build a Project History

Every project uses a schedule to communicate its milestones and to guide development efforts. This provides a means to monitor progress. It is crucial to create a schedule that is both realistic and accurate. There are many documented techniques to scientifically do this. These include estimation techniques such as Constructive Cost Mode (COCOMO), Delphi Techniques, and Gantt Charts.

When I was faced with developing a schedule for my last project, COCOMO was suggested as a useful technique. But COCOMO requires parameters such as lines of code, which were not at my disposal. Past performance also might have been a useful predictor, but most of the project team was new—to both each other and the technologies. So I decided to build a project history, albeit a brief one. The team worked without a schedule for about three months. Throughout this period, we closely monitored and recorded progress on assigned tasks. Both the team members and I gained a sense of each person's capabilities, and we based our schedule on this knowledge. I met with team members to review their assigned tasks and to estimate how long each task would take. We compared performance to these estimates on a weekly basis. Within a few months, team members could predict their progress with precision.

There were other benefits derived from this schedule-building technique. The team became intimately aware of the schedule and regularly consulted it. Also, the schedule had buy-in from all members because the team built it. As a result, motivation to achieve milestone dates was extremely high.

Minimize Meetings

In the life of a project, it is a rare day that does not include at least a few meetings. No matter how justified their purpose, meetings tend to steal valuable time from designing and developing a product, which is the real business at hand. Most team members would rather be doing their "real" work and regard meetings with disdain. To combat this bombardment of meetings, one solution is obvious: minimize their number.

This is not a trivial feat. Gathering requirements, participating in design and code inspections, attending relevant briefings, and taking part in status reviews are essential software project activities. I handled this challenge by requiring only a small subset of the entire team at different meetings. For instance, inspections included only the people necessary to ensure coverage in the areas of databases, programming languages, logic, or quality assurance. Sometimes a desk review took the place of an inspection. A few team members had dedicated roles; I designated one to be the customer interface and he represented the team at all requirements meetings.

The exception to this policy is project status reviews. Valuable information-sharing and coordination of tasks occurs at these reviews, so attendance by all team members should be mandatory.

Keep the Team Satisfied

The magic bag of project management tricks amounts to naught without the team's dedication and enthusiasm. These people put in long, hard hours to get a product out the door. The project manager must create a stress-free, positive work environment. Techniques that foster such an atmosphere include showing appreciation, injecting humor whenever possible, and empowering team members.

Project managers should take every opportunity to show their appreciation. The power of cash awards is undeniable, yet these may be unavailable for fiscal or contractual reasons. For teams that consist primarily of contractors for whom cash awards are not available, another way must be found to inform their companies of their superior efforts. At significant milestones, I awarded individuals letters of appreciation and sent a copy to their supervisors. In all cases, the employees and their companies were delighted to receive this recognition.

A little humor goes a long way and should be dispersed in large doses. When an early prototype neared completion, software samples from each team member were analyzed by the Software TestWorks tool, which rates programming style and performs coverage analysis. Much to my delight, all code received high marks. To celebrate this achievement, I awarded the programmers a mock Certificate of Excellence for their efforts. Another light moment occurred during testing when the team was on an emotional roller coaster. To alleviate the tension, I decided to recognize the person who was responsible for the hundredth software discrepancy. Everyone eagerly anticipated this event, and when it finally occurred, I presented the team member with a token of appreciation. Although work continued uninterrupted, these light moments lifted the cloud of stress.

Empowering team members reaps many benefits. It provides them with ready access to all the information they need to do their jobs. Within well-defined boundaries, I allowed developers to directly contact customers and vendors when the situation called for it. Not only did this free me for other activities, but also fostered a trusting environment in which the team felt both unfettered and motivated.

Conclusion

In today's pressure-cooker environment, projects need all the help that can be mustered. Following a cookbook approach to project management probably is not the best recipe for success. Leaders must use every technique at their disposal to achieve their project goals. The tools presented in this article are meant to complement those usually found in courses and texts. Project managers need to select those tools with which they feel the most comfortable, while remembering that project management is as much an art as it is a science. Keeping more human concerns in mind will help projects overcome challenges and attain success. ◆

About the Author

Janice Strauss has been employed at the National Security Agency as a senior computer scientist for more than 13 years. She has worked in a variety of positions, most recently as a project manager. She is also actively involved in software improvement initiatives within her current organization. These have included leading a Requirements Management Technical Working Group as well as initiating a Software Process Information Exchange group, which provides a forum to trade development tools, techniques, and best practices.

National Security Agency
9800 Savage Road
Fort Meade, MD 20755
Voice: 301-688-0994
E-mail: gusstr@erols.com

Report from STC '98

The Software Technology Conference (STC), sponsored by the U.S. Air Force, Army, Navy, and Marine Corps, and Defense Information Systems Agency, has successfully reached another milestone, completing its tenth annual conference April 19-23 in Salt Lake City, Utah. This year, more than 3,300 people from 16 nations met to exchange information, gather ideas, and draw from presentations by leading experts in software and information technology. The conference theme, "Knowledge Sharing – Global Information Networks," was likewise reflected in the displays from more than 300 vendors in the Salt Palace Convention Center Exhibition Hall and during vendor presentations.

Defense and industry leaders and other professionals agree, "Outstanding conference! ... STC sets the pace." Dr. Helmut Hellwig, deputy assistant secretary for science, technology, and engineering, Office of the Secretary of the Air Force for Acquisition, said at the conference, "We must dedicate ourselves to partnerships of people and organizations in government, industry, and academia. This will enable us to continue to manage acquisitions within the resources available and will also enable industry to make use of its past performance record, experience in the software domain or product line, and mature software development process. This is the tenth year of the annual Department of Defense Software Technology Conference. ... The conference provides a very unique opportunity for government, industry, and academia to form those partnerships vital to achieving software acquisition success. These partnerships are vital to providing American war fighters the right information in the right place, at the right time. The conference also provides a time for professional development, as attendees have the opportunity to learn more about the many faceted disciplines of software and information acquisition and engineering. Both partnerships and professional development are important aspects of ensuring our forces, industry, and country are prepared for the challenges that lie ahead in the new millennium."

Next year's conference will continue this tradition and set the stage for software and information professionals as we prepare to enter the new millennium. The theme for STC '99 is "Software and Systems for the Next Millennium." The conference co-sponsors look forward to seeing everyone May 2-6, 1999.

Dana Dovenbarger
Conference Manager
dovenbad@software.hill.af.mil

Major Causes of Software Project Failures

Lorin J. May
CROSSTALK Associate Editor

Most software projects can be considered at least partial failures because few projects meet all their cost, schedule, quality, or requirements objectives. Failures are rarely caused by mysterious causes, but these causes are usually discovered post-mortem, or only after it is too late to change direction. This article is based on interviews with software consultants and practitioners who were asked to provide "autopsies" of failed projects with which they have been acquainted. Although not a comprehensive compilation of failure causes, this article outlines several areas that should demand your attention.

A few years ago marked the rollout of what could have been called a Titanic of military projects, except the original Titanic was ahead of schedule when it sank. Hundreds of millions of dollars over budget and years behind schedule, the first phase of this huge military system was finally "tossed over the wall" and over the top of a network of separate programs used by thousands of practitioners. Although long hampered by quality problems, big hopes were again riding on the system once it passed acceptance testing.

The intended users refused to use the system. It lacked features they said were essential to their jobs while requiring steps they considered unnecessary or burdensome. The project eventually died a visible, painful death amid litigation and congressional inquiries.

This failed project was not atypical of chronic problems in the software industry. According to the Standish Group [1], in 1995, U.S. government and businesses spent approximately $81 billion on canceled software projects, and another $59 billion for budget overruns. Their survey claimed that in the United States, only about one-sixth of all projects were completed on time and within budget, nearly one third of all projects were canceled outright, and well over half were considered "challenged." Of the challenged or canceled projects, the average project was 189 percent over budget, 222 percent behind schedule, and contained only 61 percent of the originally specified features.

Other studies have likewise concluded that failure is rampant, although not necessarily to the same degree. One reason for the varied conclusions is that most failed projects are never studied—even by the organization that experienced the failure. Having wasted so much on a fruitless venture, few organizations will invest more time or money to collect and analyze additional data, whereas any data that had been collected may be massaged or hidden to protect careers or reputations. Thus, information about project failures often relies heavily on subjective assessments. This article is no exception.

For this article, a failure is defined as any software project with severe cost or schedule overruns, quality problems, or that suffers outright cancellation. It is based on interviews with practitioners and consultants who were asked to describe the causes of software project failures with which they have been acquainted. If there is anything notable about the interviewees' diagnoses, perhaps it is that many of these problems have been documented for years, but somehow they keep cropping up. Also worth noting is that most of the failure causes mentioned originate before the first line of code has been written. The failure causes are listed in no particular order.

Poor User Input

Although the Titanic project mentioned earlier was riddled with problems, it ultimately failed because the system did not meet user needs. According to Paul Hewitt, a consultant with the Software Technology Support Center (STSC), the acquirers and developers of this system had received most of their requirements from higher-level supervisors and so-called "users" who were not regularly using the existing system. Although "not invented here" syndrome contributed to the system's eventual lack of acceptance, the bottom line is that the system was inadequate for its environment.

By contrast, Hewitt has observed successful programs in which "end users and developers [were] working together in the same cubicle." Although this is not always possible, Hewitt said projects are likely headed for trouble unless informed end users are giving meaningful input during every phase of requirements elicitation, product design, and building. The input needed by these users has less to do with issues like screen layouts than with how the system would be used in the field, according to Michael Allen Latta, chief executive officer of Ohana Technologies Corp. in Lafayette, Colo. He said the user should be asking, "How do I use it over time? Does it provide the right tools? What do I put into it, and what do I get out?"

However, there can also be problems if the users are too close to the requirements. Shari Lawrence Pfleeger, president of Systems/Software in Washington, D.C., had just started consulting on a large federal system acquisition when she started to study its requirements, which were supposedly "clean" due to the input of highly knowledgeable users. Even without any prior understanding of the system or its field environment, Pfleeger needed only a few hours to see that the requirements were full of hidden assumptions and conflicts.

"[The users] didn't think of the consequences of what they were requiring," she said. "They assumed that how things were done in the past was how they would always be done in the future."

The users assumed the elicitors understood more than they did about the users' jobs, but this was not entirely the users' fault. *All* involved parties, including the developers, must understand the business of the other parties. This need continues throughout development process. Without this understanding, the parties "don't even know what questions to ask," Pfleeger said, and important issues fall between the cracks.

Stakeholder Conflicts

A few years ago, a major airline, rental car company, and some hotel chains created an incentive plan to give customers frequent flier-type points to "cash in" for any of the participating companies' services. They commissioned a complex software system to track points and compensation. Sometime later, the software developers needed some clarifications, i.e., with input A, does the system choose X, Y, or Z? The stakeholders could not agree on the answers. Forced to acknowledge deep incompatibilities among their business interests, the system was canceled in an expensive, litigious failure of the entire enterprise.

The stakeholders had worked under "the illusion that everyone was going to get everything that they wanted," explained Tom DeMarco, principle of the Atlantic Systems Guild. They "papered over their differences" rather than going through conflict resolution in the early stages. Their differences were exposed by the developers because "coders cannot make an ambiguous system."

Stakeholder conflicts can play many different roles project failures. For example, "some projects are ultimately canceled because people don't like each other," said Capers Jones, chairman of Software Productivity Research, Inc.

Other projects fail because the developers do not know who the "real" stakeholders are, according to Ed Yourdon, chairman of the Cutter Consortium. Yourdon worked with a large mutual fund company that had been working on a $300 million software system. The developers had been working closely with the information technology vice president, who was perceived to be the primary stakeholder for the system. When the system ran into some problems, it drew the attention of the chief executive officer, who turned out to be the real stakeholder in the system even though he had not previously been involved with it. After seeing the involved risks, he immediately withdrew his support for the system.

"No one bothered to ensure that he was going to support it," Yourdon explained. "No one made him aware of problems while it was being developed." Yourdon says many projects fail because the project leaders do not have a sense of who will ultimately declare whether a project is a success or failure, and then they are "blindsided." He said the true stakeholders need to hear good and bad news in "small pieces" rather than in "one chunk."

Other projects, especially smaller projects within larger projects, never go anywhere because the internal stakeholders never agree on priorities. Watts Humphrey, a fellow at the Software Engineering Institute, calls these "pretend projects," meaning a few developers work on them half time or quarter time, and nothing is ever delivered.

"They are kidding themselves that they are working on [these projects]," Humphrey said. "No one can work quarter time on a project. ... They haven't faced the need as a management team to decide what they are really going to do with it. They need to put real resources on it" rather than merely pretend the project is under way.

Vague Requirements

Mariea Datiz, president of Peripheral Visions in Houston, Texas, learned a hard lesson about what happens when a project is started while the requirements are nebulous. The U.S. division of an oil company hired Datiz's company to create the "first draft" of a program so that they could impress their European counterparts and justify further funding. But the oil company officials only had a general idea of what the program was to do and tried to revise and refine their ideas while Datiz's company was working on the program.

"For every step we would take, we'd go three backward," Datiz said. "We would start down one path and then have to stop and go down another." Project cost and quality quickly went out of control, her company was blamed, and she lost the contract to finish the job. Like many failed projects, the scope had not been narrowed enough at the outset to have led to any reasonable chance for success.

One obvious solution is to establish a reasonably stable requirements baseline before any other work goes forward. But even when this is done, requirements will still continue to creep. "You can't design a process that assumes [requirements] are stable," advises Humphrey. In virtually all projects, there will be some degree of "learning what the requirements really are while building the product," he said. Projects could be headed for trouble if architectures and processes are not change-friendly, or if there are poorly established guidelines that determine how and when requirements can be added, removed, and implemented—and who will shoulder the cost of the changes.

Poor Cost and Schedule Estimation

It is unfair to call a project a failure if it fails to meet budget and schedule goals that were inherently unattainable. Like all engineering endeavors, every software project has a minimum achievable schedule and cost. Fredrick Brooks summarized this law in *The Mythical Man Month* [2] when he stated, "The bearing of a child takes nine months, no matter how many women are assigned." Attempts to circumvent a project's natural minimum limits will backfire.

This problem occurs any time someone "makes up a number and won't listen to anyone about how long other projects took," said Jones. According to DeMarco, projects are often intentionally underbid because of the "attitude that putting a development team under sufficient pressure can get them to deliver almost anything."

The opposite is what usually happens. For example, if a program should realistically take five programmers one year to complete, but instead you are given four programmers and eight

months, you will have to skimp on design time and on quality checks to reach project milestones.

"Cutting a corner that undermines the entire foundation of the project is not cutting the corner," states Robert Gezelter, a software consultant in Flushing, New York. "There will be heavily disproportionate costs downstream." Skimping leads to weak designs, dramatically higher defect densities, much more rework, and virtually endless testing. In the end, the project will cost more, take longer, and have worse quality than would have been possible if a realistic schedule and budget had been followed.

According to Jones, this problem can be easily remedied. Several estimation tools on the market can combine numerous variables to provide realistic estimates within a few hours [3], even at the early critical decision-making junctures—before requirements are firm.

Skills that Do Not Match the Job

Decades ago, Morris Dovey, information director for Check Control, Inc. in West Des Moines, Iowa, worked on major government software contracts before becoming so frustrated he decided to never work with government contracting again.

"It was being made artificially difficult," Dovey said. The technologists had to endure what he considered avoidable delays and mistakes because "decisions were being made by people with no technical expertise in the area" but had all the authority.

Latta warns that managers can perform poorly if they lead projects that do not match their strengths. "Projects dealing with high technology need managers with solid technical skills," Latta advises. In such projects, authority must reside with people who understand the implications of specific technical risks.

However, the best technologists are not necessarily always poised to be the best managers. "The skill set for management and programming are disjoint," Jones observed. The larger the project, the more need there is for people with excellent planning, oversight, organization, and communications skills; excellent technologists do not necessarily have these abilities.

Skill-driven challenges are not limited to management. Poor developers can sap productivity and make critical, expensive errors. Generalists can also poorly perform duties better left to specialists, such as metrics experts or testers.

The solution to skill-driven challenges is easy to define but difficult and expensive to accomplish: Attract and retain the most highly skilled and productive people. "Knowledge is money," noted Tom Pennington, senior network manager for The MIL Corporation in Arlington, Va. However, there is an eventual payback. Pennington believes a team made up of higher-paid people with the right specialized skills is worth far more per dollar to an organization than a group of lower-cost people who need weeks or months of fumbling through a new process or technology before they can start being productive.

"You get what you pay for," Datiz echos. "You'll also pay for what you get."

Jones advises that "if you can't get the best 'techies,' get the best managers." He said good managers can often get above-average results from average employees, whereas great employees can have much of their potential squandered by mediocre management.

Hidden Costs of Going "Lean and Mean"

DeMarco believes project managers and technologists are often unfairly blamed for problems caused by people "two levels higher." He believes managers and technologists are generally competent and getting better every year, but they are "goaded" into overtime work because of "the 1990s stupid flirtation with lean and mean"—cutting jobs and expecting the same work with fewer people and less money, whether such a feat is possible or not. DeMarco says the the often-intentional "dishonest pricing" of projects is often off by a factor of two or four or more, requiring never-before-seen levels of performance.

"Any failure will be viewed as a direct result of underperformance," he charges, even though underperformance is "not even a significant factor" in the failure of most projects. Instead, he says, the failed projects simply had goals that were inherently unattainable.

Humphrey has observed a different "lean and mean" problem. In many "downsized" organizations, he says, developers are doing their own expense accounts, clerical work, software updates, and other duties—and at a higher labor rate and with less skill than could be performed by support specialists.

He estimates that many software developers are spending half their work hours slowly plodding through tasks that have nothing to do with developing software. "Software people are very unskilled clerks," he said. "It's an enormous productivity issue."

Failure to Plan

Humphrey took charge of commercial software development for IBM at a point when the company was taking too long to finish projects and was missing all its announced deadlines. "People were working hard, but no one had plans ... because no one required them to make plans," Humphrey recalls. In response, he required that a detailed plan be developed before any release date was announced. For the next two and one half years, the division never missed an announced date.

"If software developers built bridges, we'd show up at the site with some scrap iron and say, 'let's start building!'" quipped Reuel Alder, a manager at the STSC. Alder agrees that inadequate planning is a major reason software projects spin out of control.

Humphrey said project managers often do not plan because "any plan they put together won't meet the [desired release] date, so they can't plan." Even though detailed planning saves an enormous amount of time in the long run, Humphrey says many other managers and developers believe it to be unnecessary. "They think time spent on things like planning, design, requirements, and inspection gets in the way of real work, which is coding and testing," he said. "This comes from the view of programming that the issue is to get the software out the door. But there's a difference between speed and progress."

"We need a lot fewer heroes," adds Gezelter. He believes organization "heroics" would frequently be unnecessary if projects had been properly planned. "We keep rewarding people for charging off on suicide missions," he said.

Communication Breakdowns

When Pfleeger was asked to consult on a large project that was in trouble, she asked the managers to develop a process model for the project. She did not necessarily want the model for her own use, but wanted the managers to talk to the developers. Once they did, they realized the project had gotten so large that the same code was being tested by two teams that did not know the other existed.

Such problems are common on large projects, especially if people are working at different sites. In many troubled projects, "there isn't one person who has an overview of the whole project," she said. Especially on large projects, Pfleeger advises that additional time be taken periodically to have everyone in every position learn the big picture. "The people working on the pieces need to know how their one piece fits into the entire architecture."

Poor Architecture

Pfleeger says an example of flexible architecture is the Patriot missile used during the Gulf War. It was not designed to intercept scud missiles, but the software was able to be reconfigured to support the new function. On the other end of the flexibility spectrum was a security program created to protect sensitive word-processing documents. Everything worked well for a few months until the operating system was updated. The word-processing programs still worked, but the security program became useless and unfixable because much of its code was tied to operating system features that were dropped in the new system.

"People didn't think ahead about what was likely to change," Pfleeger said. Architecture must allow for organization and mission changes.

Gezelter said software developers often build with no more forethought than the man who built a beautiful boat in his workshop and then could not get it out the door. "If you do [architecture] right, no one will ever realize it," he said. "But if you do it wrong, you will suffer death by a thousand cuts. Bad choices show up as long-term limitations, aggravation, and costs."

Gezelter suggests viewing software architecture like house-building: "Plumb" and "wire" for features and additions you have not thought of yet. Then, when unanticipated needs or business changes arise, you can add or modify without performing the software equivalent of "ripping apart the walls and rebuilding them again."

Late Failure Warning Signals

Does the following scenario by Yourdon seem familiar? A schedule and budget are determined "by edict by people you were afraid to say no to," and it is politically unwise either to say or show the estimate is far from achieveable. All your early milestones involve diagrams, designs, and other documents that do not involve working code. These and other project milestones then go by more or less on schedule—at least as far as upper management can tell—and testing starts more or less on time. Not until the project is a few weeks from deadline does anyone dare inform the "edict makers" that at the current defect detection rate, the project will not be completed even close to its deadline.

"Nobody seems to acknowledge that disaster is approaching," Yourdon said, even among people who sense there is a problem. "There is no early warning signal." Until more organizations abandon waterfall-style development in favor of processes that demand early working code or prototypes, he says this scenario will continue to be familiar.

Yourdon says the above problem is also extremely common with year 2000 work. He believes many year 2000 conversion teams, if they were allowed, would say of their current situation: "Within this limited time and pitiful budget and understaffed team, sure, we can deliver it on time—with a million bugs in it."

In a perfect world, lower-level people could convince upper-level managers that their edicts are unworkable before the project got under way. But until this happens, Yourdon says development cycles need to be adopted that allow you, at the earliest possible moment, to "provide evidence that [the project] is or is not working."

Conclusion

Other causes of failure could be added ad nauseam, but the existence of additional factors is not the point. As Jones noted, "There are myriad ways to fail. … There are only a very few ways to succeed." [3] The factors of successful project management have been documented for years—they merely need greater attention. But if this article has helped serve as a reality check for your project, it will have served its purpose. If you violate any of the principles noted by the consultants and practitioners in this article, you should not expect to succeed in spite of yourself. ◆

About the Author

Lorin J. May is an editor and columnist for *CROSSTALK: The Journal of Defense Software Engineering.* He is employed by NCI Information Systems, Inc., under contract to *CROSSTALK* at the Software Technology Support Center. He was previously an editor for two book publishers and was a part-time freelance writer. He has a bachelor's degree in journalism from Weber State University in Ogden, Utah.

OO-ALC/TISE
7278 Fourth Street
Hill AFB, UT 84056-5205
Voice: 801-777-9239 DSN 777-9239
Fax: 801-777-8069 DSN 777-8069
E-mail: MayL@software.hill.af.mil

References

1. The Standish Group, "Chaos," 1995, http://www.standishgroup.com/chaos.html.
2. Brooks Jr., Frederick P., *The Mythical Man-Month* (20th Anniversary Edition), Addison-Wesley, Reading, Mass., 1995.
3. Jones, Capers, *Patterns of Software Systems Failure and Success,* International Thompson Computer Press, Boston, Mass., 1996.

Project Management Tools and Software Failures and Successes

Capers Jones
Software Productivity Research, Inc.

The construction of large software systems is one of the most hazardous activities of the business world. The failure or cancellation rate of large software systems is over 20 percent. Of the large systems that are completed, about two thirds experience schedule delays and cost overruns that may approach 100 percent. About the same number are plagued by low reliability and quality problems in the first year of deployment. Yet, some large systems are finished early, meet their budgets, and have few if any quality problems. How do successful projects differ from projects that fail? Better project management and better quality control are the most important differences between success and failure in the software world.

Software development is a troubling technology. Software is highly labor-intensive, and as a result, large software projects are among the most expensive undertakings of the 20th century. Large software systems cost far more to build and take much longer to construct than the office buildings occupied by the companies that have commissioned the software. Extremely large software systems in the 100,000 function point size range can cost more than building a domed football stadium, a 50-story skyscraper, or a 70,000-ton cruise ship.

Consider what the phrase "large systems" means in the context of six different size plateaus separated by an order of magnitude for each plateau. Size is expressed in terms of function points, a widely used synthetic metric based on five external attibutes of software applications: inputs, outputs, inquiries, logical files, and interfaces. The average number of C statements found within the typical function point is provided as a point of reference.

One Function Point (125 C Statements)

There are few software applications of this size except small enhancements to larger applications or minor personal applications. The schedules for such small programs are usually only from a day to perhaps a week.

10 Function Points (1,250 C Statements)

This is the typical size of end-user applications and also a tremendously frequent size plateau for enhancements to existing software. Development schedules are usually less than one month.

100 Function Points (12,500 C Statements)

This size is heavily populated with enhancements to existing applications. It is also the practical upper limit of end-user applications. There are few stand-alone applications of this size in 1998, but 10 years ago there were a number of DOS applications in this size range, such as early BASIC interpreters. However, there are many features of larger applications that approximate this size. Development schedules are usually less than six months. Individual programmers can handle applications of this size, although technical writers and other specialists may be involved, too.

1,000 Function Points (125,000 C Statements)

This size range exceeds the capabilities of end-user development. This is a fairly common entry-level size range for many commercial and internal Windows software applications. It is also a common size range for in-house client-server applications. Schedules for software projects of this size are usually longer than 12 months. In this size range, the volume of specifications and user documentation becomes a significant contributor to software costs.

Quality control also is a major requirement at this size range. Applications of this size range require development teams of up to 10 staff members, since individual programmers cannot usually handle the volume of code and other deliverables. Specialties such as quality assurance, technical writing, and database administration are often represented on the development team. With team development, issues of system segmentation and interfaces among components become troublesome.

10,000 Function Points (1,250,000 C Statements)

Applications of this size are usually termed "systems" because they are far too large for individual programs. This size range is often troubled by cost and schedule overruns and by outright cancellations. Development teams of 100 or so are common, so communication and interface problems are endemic.

Software schedules in this size plateau run from three to more than five years, although the initial planning for applications of this size range tends to naively assume schedules of 18 months or less. The volume of paperwork in terms of plans, specifications, and user manuals is so large that production of documents is often more expensive than the source code. Because defect levels rise with application size, formal quality control including pre-test inspections are

necessary for successful completion. Configuration control and change management also are mandatory for this size plateau.

100,000 Function Points (12,500,000 C Statements)

Applications that approach 100,000 function points in size are among the most troubling constructs of the 20th century. This is roughly the size range of Microsoft's Windows 95 product and also IBM's MVS operating system. This is also the size range of major military systems.

Software development schedules for systems of this size are usually from five to more than eight years, although the initial development plans tend to assume 36 months or less. Development teams number in the hundreds, often in multiple locations that may even be in different countries. Communication problems are rampant. Paperwork and defect removal operations will absorb the bulk of development costs. Formal configuration control and change management are mandatory and expensive for this size plateau.

Using these six size ranges, Table 1 shows the approximate frequency of various kinds of outcomes, ranging from finishing early to total cancellation. Table 1 is taken from *Patterns of Software Systems Failure and Success* (International Thomson Computer Press, 1996).

As can easily be seen from Table 1, small software projects are successful in the majority of instances, but the risks and hazards of cancellation or major delays rise quite rapidly as the overall application size goes up. Indeed, the development of large applications in excess of 10,000 function points is one of the most hazardous and risky business undertakings of the modern world.

Software Successes and Disasters Within Six Subindustries

There are six major subindustries within the software community that tend to follow somewhat different practices and even use different tools and programming languages. In terms of their ability to successfully build large software applications, these six subindustries in order of rank are

- Systems software.
- Outsource vendors.
- Commercial software.
- Military software.
- Management information software.
- End-user software.

It is interesting to consider why there are variances among these industries in the ability to complete large software projects. These six categories are the most common types of software development projects in North America, South America, Europe, Africa, India, the Middle East, and the Pacific Rim.

Systems Software

This category refers to applications that control physical devices such as operating systems, navigation and flight control, telecommunication systems, process control systems, automotive fuel injection, medical instruments, and the like. The systems software community is concerned with software that operates large and complex physical devices. If quality is not excellent, then the devices may fail during use; therefore, the systems software community has learned the hard way that careful quality control is on the critical path. The systems software community, overall, has the best track record for building large software applications. This community also has the best quality control and the best suites of quality control tools.

Outsource Vendors

These are companies such as Andersen Consulting, Computer Sciences Corporation, Electronic Data Systems, IBM's Integrated Systems Solutions, and a number of others. These companies build software under contract for their clients. As a class, the outsource vendors are often better equipped and better trained than the clients they serve. This is not always true, but if it were not true, fairly often, the outsource business would fail. The outsource community often has highly sophisticated project management and quality control tool suites available, significant amounts of reusable material, and highly trained personnel.

Military Software

These are applications constrained to follow various military standards such as the older DOD-STD-2167A standard or the newer MIL-STD-498. Military applications that control weapons systems tend to resemble civilian systems software projects in terms of the emphasis on careful planning and quality control.

The military and defense community is not in reality bad at building large systems, but there is a major problem in this domain. Military standards are so complex and baroque that the productivity of defense applications is lower than any other software subindustry. The reason for low productivity has nothing to do with coding or technical work. Military standards trigger such enormous volumes of paperwork that there are roughly 400 English words created for every Ada statement on military software projects. The volume of paperwork on military software projects is almost three times that of comparable civilian projects of the same size.

Table 1. *Software project outcomes by size of project.*

	PROBABILITY OF SELECTED OUTCOMES				
	Early	On Time	Delayed	Canceled	Sum
1 FP	14.68%	83.16%	1.92%	0.25%	100.00%
10 FP	11.08%	81.25%	5.67%	2.00%	100.00%
100 FP	6.06%	74.77%	11.83%	7.33%	100.00%
1,000 FP	1.24%	60.76%	17.67%	20.33%	100.00%
10,000 FP	0.14%	28.03%	23.83%	48.00%	100.00%
100,000 FP	0.00%	13.67%	21.33%	65.00%	100.00%
Average	5.53%	56.94%	13.71%	23.82%	100.00%

	Systems Software	Military Software	MIS Software	Outsource Software	Comm. Software	End-User Software	Average
1 FP	99.00%	98.00%	98.00%	98.00%	99.00%	95.00%	97.83%
10 FP	96.00%	93.00%	95.00%	97.00%	98.00%	75.00%	92.33%
100 FP	88.00%	84.00%	86.00%	88.00%	89.00%	50.00%	80.83%
1,000 FP	75.00%	65.00%	68.00%	74.00%	75.00%	5.00%	60.33%
10,000 FP	54.00%	38.00%	30.00%	47.00%	35.00%	0.00%	40.80%
100,000 FP	28.00%	15.00%	5.00%	24.00%	10.00%	0.00%	18.40%
Average	73.33%	65.50%	63.67%	71.33%	67.67%	37.50%	65.09%

Table 2. *Probability of on-time software delivery in six subindustries.*

Commercial Software

This refers to the high-volume shrink-wrapped packages by companies such as Borland, Computer Associates, and Microsoft. Until recently, this subindustry did not build many large applications, so they have had some catching up to do. As the size of personal computer software packages approaches the size of mainframe software packages, the commercial vendors have had to increase their project management tools and methods, strengthen quality control, and in general, imitate the successful pattern of the systems software domain.

Management Information Systems (MIS)

This refers to the internal applications companies build for their own use: accounting systems, payroll systems, insurance claims handling, banking and financial systems, etc. The MIS community does not have a particularly good track record when it comes to large systems. Often, the MIS community lags in quality control and testing technologies compared to the other communities. However, project management tools for MIS companies are now increasing in number and capability.

End-User Software

This refers to applications built privately by people for their own use, which in the context of this article means applications used for business or professional purposes, not games or home applications. Although tools such as Visual Basic, Realizer, spreadsheets, and SAS have expanded the capabilities of the end-user community, there is still a low upper limit to the sizes of applications that end users can construct. About 100 function points is the practical upper limit and 1,000 function points is the current maximum size of end-user applications.

Probabilities of On-Time Software Delivery, Cancellations, or Delays

The first summary topic of interest is the probability that software projects will be finished on time, using the initial schedule estimate derived during requirements as the basis of the comparison. Table 2 shows the on-time rates but needs some explanation first. There is an anomaly in the data because there are no end-user applications larger than 1,000 function points; therefore, the 0 percent values in the end-user column are excluded from the average values. Also, some projects finish early, but these are included in the on-time percentages. The probability of an early finish for 10,000 function points or larger is approximately 0 percent.

As can be seen, small software projects are comparatively well controlled within all six subindustries. As the overall size ranges grow larger, delays and cancellations become much more common and also more severe. On the whole, the systems software community and the outsource community have the best results with systems in the 10,000 to 100,000 function point range; the military domain comes in third place.

Probability of Termination

The next topic of interest is the probability that a project will be terminated prior to completion. This is among the most severe risks we face in software—only termination with accompanying litigation is more disastrous. Table 3 shows the probabilities of software project terminations for the various subindustries.

To illustrate our failures on an intuitive level, consider the following analogy: If building construction had the same ratio of cancellations as software, more than half the office buildings in the world larger than 30 stories tall would be abandoned before completion. The average height of buildings in New York City would be only three stories, and there would be no skyscrapers.

None of the six domains have fully mastered the ability to construct large software systems without a significant risk of termination or cancellation. However, the systems software community and the outsource community have the best track record for large systems, with the military software community coming in third. The information systems community fails repeatedly for large systems. The commercial software world is not particularly good at the large system plateau—though it is getting better—and end users cannot do large systems.

Table 3. *Probability of software project termination in six subindustries.*

	Systems Software	Military Software	MIS Software	Outsource Software	Comm. Software	End-User Software	Average
1 FP	0.10%	0.10%	0.10%	0.10%	0.10%	1.00%	0.25%
10 FP	1.00%	2.00%	1.00%	1.00%	2.00%	5.00%	2.00%
100 FP	5.00%	7.00%	6.00%	6.00%	5.00%	15.00%	7.33%
1,000 FP	12.00%	15.00%	17.00%	14.00%	9.00%	65.00%	22.00%
10,000 FP	25.00%	33.00%	45.00%	40.00%	45.00%	100.00%	48.00%
100,000 FP	40.00%	55.00%	80.00%	45.00%	70.00%	100.00%	65.00%
Average	13.85%	18.68%	24.85%	17.68%	21.85%	47.67%	24.10%

	Systems Software	Military Software	MIS Software	Outsource Software	Comm. Software	End-User Software	Average
1 FP	0.90%	1.90%	1.90%	1.90%	0.90%	4.00%	1.92%
10 FP	3.00%	5.00%	4.00%	2.00%	0.00%	20.00%	5.67%
100 FP	7.00%	9.00%	8.00%	6.00%	6.00%	35.00%	11.83%
1,000 FP	13.00%	20.00%	15.00%	12.00%	16.00%	30.00%	17.67%
10,000 FP	21.00%	29.00%	25.00%	13.00%	20.00%	0.00%	11.20%
100,000 FP	32.00%	30.00%	15.00%	31.00%	20.00%	0.00%	16.60%
Average	12.82%	15.82%	11.48%	10.98%	10.48%	14.83%	10.81%

Table 4. *Probability of schedule slip by more than 25 percent in six subindustries.*

Probability of Schedule Overrun

The next topic of interest is the probability that a software project will eventually be finished but will run later than anticipated by a significant amount (a 5 percent slip is noticeable, more than 10 percent is painfully costly, and a 50 percent slip is a catastrophe). The initial estimate developed during requirements is the starting point. Table 4 shows slippage probabilities for the six subindustries.

Here, too, the low end of the software size spectrum is generally trouble-free and under full control. As the size range gets larger, delays and cancellations become much more common. A contributing factor to both delays and cancellations also grows with size: The probability of "creeping user requirements." The average growth of unplanned, unanticipated requirements is about 1 percent to 2 percent per month during the design and coding phases of typical software projects, although the upper range of requirements creep can exceed 10 percent in a single month.

Any of the six domains can build small software projects with a good probability of success. At the upper end, no domain is fully capable. However, the systems software world, the large outsource contractors, and the military domains are the most experienced with large applications and therefore have somewhat better probabilities of succeeding.

Project Management Tools Used on Successful Software Projects

One of the newer uses of the function point metric is to evaluate the completeness of various kinds of software tool suites. This approach can clearly reveal some of the critical differences between successful software projects, average projects, and total failures.

It is obvious to consultants who spend much time with large systems and large corporations that manual methods are not adequate for cost estimation, schedule planning, or quality prediction. The best-in-class organizations may have more than 10 times the quality tool capacities and more than 30 times the project management tool capacities than the organizations that fail with software.

Interestingly, there may be little if any difference in the capacities of software engineering tool suites. Both successful and unsuccessful companies tend to have in the range of 30,000 to perhaps 50,000 function points of software engineering and development tools. The difference between companies that succeed and those that do not is that the former employ effective project management tool suites whereas the latter generally do not. Table 5 identifies the typical patterns of project management tools noted on leading, average, and lagging software projects.

As shown, the lagging projects tend to be essentially manual for most project management functions. The leading projects deploy a notable quantity of quality control and project management automation. Leading projects tend to use more than 16 times the project management tool capacities of lagging projects in terms of function points. In terms of numbers of project management tools deployed, there is about a 6-to-1 ratio between the leading and lagging projects.

The presence of a suite of project management tools is not, by itself, the main differentiating factor between successful and unsuccessful software projects. The primary reason for the differences noted between lagging and leading projects is that the project managers who use a full suite of management tools are usually better trained and have a firmer grasp of the intricacies of software development than the managers who lack adequate management tools.

Bringing a large software project to a successful conclusion is an extremely difficult task filled with complexity. The managers who can deal with this complexity recognize that some of the cost and resource scheduling calculations exceed the ability of manual methods. Managers on failing projects, on the other hand, tend to have a naive belief that project planning and estimating are simple enough to be done using rough rules of thumb and manual methods.

Summary and Conclusions

Software is intangible, but the schedules and cost estimates for software can be highly tangible. Software projects are still subject to the basic laws of manufacturing, and software needs to be placed on a

Table 5. *Numbers and size ranges of project management tools (size data expressed in terms of function point metrics).*

Project Management	Lagging	Average	Leading
Project planning	1,000	1,250	3,000
Project cost estimating			3,000
Statistical analysis			3,000
Methodology management		750	3,000
Year 2000 analysis			2,000
Quality estimation			2,000
Assessment support		500	2,000
Project measurement			1,750
Portfolio analysis			1,500
Risk analysis			1,500
Resource tracking	300	750	1,500
Value analysis		350	1,250
Cost variance reporting		500	1,000
Personnel support	500	500	750
Milestone tracking		250	750
Budget support		250	750
Function point analysis		250	750
Backfiring: LOC to FP			750
Function point subtotal	1,800	5,350	30,250
Number of tools	3	10	18

firm engineering basis by the end of the 20th century.

Project managers are the primary key to software project success and failures. To a large degree, the sophistication or lack of sophistication of the project management tool suite will determine whether software projects will succeed, experience major cost and schedule overruns, or fail. ◆

About the Author

Capers Jones is an international consultant on software management topics and chairman of Software Productivity Research, Inc. (SPR) in Burlington, Mass. He began his software career as a programmer in the Office of the Surgeon General, Washington, D.C. Prior to becoming chairman of SPR, he worked at the Crane Company, IBM, and was assistant director of programming technology at ITT Corporation's Programming Technology Center in Stratford, Conn.

Software Productivity Research, Inc.
1 New England Executive Park
Burlington, MA 01803-5005
Voice: 781-273-0140
Fax: 781-273-5176
E-mail: capers@spr.com

Suggested Readings

1. Brown, Norm, ed., *The Program Manager's Guide to Software Acquisition Best Practices*, Version 1.0, U.S. Department of Defense, Washington, D.C., July 1995.
2. Charette, Robert N., *Software Engineering Risk Analysis and Management*, McGraw-Hill, New York, 1989.
3. Charette, Robert N., *Application Strategies for Risk Analysis*, McGraw-Hill, New York, 1990.
4. DeMarco, Tom, *Controlling Software Projects*, Yourdon Press, New York, 1982.
5. DeMarco, Tom, *Why Does Software Cost So Much?*, Dorset House, New York, 1995.
6. Department of the Air Force, *Guidelines for Successful Acquisition and Management of Software-Intensive Systems*, Vols. 1 and 2, Software Technology Support Center, Hill Air Force Base, Utah, 1994.
7. Dreger, Brian, *Function Point Analysis*, Prentice-Hall, Englewood Cliffs, N.J., 1989.
8. Grady, Robert B., *Practical Software Metrics for Project Management and Process Improvement*, Prentice-Hall, Englewood Cliffs, N.J., 1992.
9. Grady, Robert B. and Deborah L. Caswell, *Software Metrics: Establishing a Company-Wide Program*, Prentice-Hall, Englewood Cliffs, N.J., 1987.
10. IFPUG Counting Practices Manual, Release 4, International Function Point Users Group, Westerville, Ohio, April 1995.
11. Jones, Capers, *Applied Software Measurement*, McGraw-Hill, New York, 2d ed., 1996.
12. Jones, Capers, *Critical Problems in Software Measurement*, Information Systems Management Group, 1993.
13. Jones, Capers, *Software Productivity and Quality Today – The Worldwide Perspective*, Information Systems Management Group, 1993.
14. Jones, Capers, *Assessment and Control of Software Risks*, Prentice-Hall, 1994.
15. Jones, Capers, *New Directions in Software Management*, Information Systems Management Group.
16. Jones, Capers, *Patterns of Software System Failure and Success*, International Thomson Computer Press, Boston, Mass., December 1995.
17. Jones, Capers, *Software Quality – Analysis and Guidelines for Success*, International Thomson Computer Press, Boston, Mass., 1997.
18. Jones, Capers, *The Economics of Object-Oriented Software*, Software Productivity Research, Burlington, Mass., April 1997.
19. Jones, Capers, *The Year 2000 Software Problem – Quantifying the Costs and Assessing the Consequences*, Addison-Wesley, Reading, Mass., 1998.
20. Kan, Stephen H., *Metrics and Models in Software Quality Engineering*, Addison-Wesley, Reading, Mass.
21. Howard, Alan, ed., *Software Metrics and Project Management Tools*, Applied Computer Research, Phoenix, Ariz., 1997.
22. Mertes, Karen R., *Calibration of the CHECKPOINT Model to the Space and Missile Systems Center Software Database*, thesis, AFIT/GCA/LAS/96S-11, Air Force Institute of Technology, Wright-Patterson Air Force Base, Ohio, September 1996.
23. Multiple authors, *Rethinking the Software Process*, (CD-ROM), Miller-Freeman, Lawrence, Kan., 1996. (This is a new CD-ROM book collection jointly produced by the book publisher, Prentice-Hall and the journal publisher, Miller-Freeman. This CD-ROM disk contains the full text and illustrations of five Prentice-Hall books: *Assessment and Control of Software Risks* by Capers Jones, *Controlling Software Projects* by Tom DeMarco, *Function Point Analysis* by Brian Dreger, *Measures for Excellence* by Larry Putnam and Ware Myers, and *Object-Oriented Software Metrics* by Mark Lorenz and Jeff Kidd.)
24. Putnam, Lawrence H., *Measures for Excellence – Reliable Software On Time, Within Budget*, Yourdon Press - Prentice-Hall, Englewood Cliffs, N.J., 1992.
25. Putnam, Lawrence H. and Ware Myers, *Industrial Strength Software – Effective Management Using Measurement*, IEEE Press, Los Alamitos, Calif., 1997.
26. Rubin, Howard, *Software Benchmark Studies for 1997*, Howard Rubin Associates, Pound Ridge, N.Y., 1997.
27. Stukes, Sherry, Jason Deshoretz, Henry Apgar, and Ilona Macias, *Air Force Cost Analysis Agency Software Estimating Model Analysis*, TR-9545/008-2, Contract F04701-95-D-0003, Task 008, Management Consulting & Research, Inc., Thousand Oaks, Calif., Sept. 30, 1996.
28. Symons, Charles R., *Software Sizing and Estimating – Mk II FPA (Function Point Analysis)*, John Wiley & Sons, Chichester, England, 1991.
29. Thayer, Richard H., ed., *Software Engineering and Project Management*, IEEE Press, Los Alamitos, Calif., 1988.
30. Umbaugh, Robert E., ed., *Handbook of IS Management*, 4th ed., Auerbach Publications, Boston, Mass., 1995.
31. Zells, Lois, *Managing Software Projects – Selecting and Using PC-Based Project Management Systems*, QED Information Sciences, Wellesley, Mass.

Coming Events

Call for Papers: First International Software Assurance Certification Conference (ISACC'99)

Theme: Early Lessons Learned and Prospects
Location: Washington, D.C.
Dates: March 1-3, 1999
Chairman: Chuck Howell, E-mail: howell@rstcorp.com
Program Chairman: Jeffrey Voas, E-mail: jmvoas@rstcorp.com
Sponsor: Software Testing Assurance Corporation, Stamford, Conn.
Conference Secretariat: Peggy Wallace
Voice: 703-404-9293
Fax: 703-404-9295
E-mail: pwallace@rstcorp.com
Internet: http://www.rstcorp.com/ISACC99

Software Cost and Schedule Estimation Course

Dates: July 13-15, l998
Location: University of California at Los Angeles
Topics: How to estimate software project cost and schedule, including metrics, case studies, productivity of developers, quality-ISO 9001 and SEI CMM, managing the estimation process, rules of thumb, why projects succeed or fail, advantages and disadvantages of widely used models, Year 2000 challenge, emerging issues, and reference sources.
Sponsor: UCLA Extension Short Course Program
Contact: Marcus Hennessy
Voice: 310-825-1047
Fax: 310-206-2815
E-mail: mhenness@unex.ucla.edu
Contact: Donald S. Remer
Voice: 909-621-8964
E-mail: remer@hmc.edu

12th Annual ASEET Symposium

Dates: July 27-30, 1998
Location: Monterey, Calif.
Presenters: Dawn Hartley (Defense Information Systems Agency), Col. Kevin J. Cogan (U.S. Military Academy), Martin C. Carlisle (U.S. Air Force Academy), Dean Hendrix (Auburn University), Ben Brosgol (Aonix), Richard Riele (AdaWorks).
Contact: Lt. Col. Drew Hamilton
Voice: 914-938-5555 DSN 688-5555
Fax: 914-938-5956 DSN 688-5956
E-mail: dj7560@exmail.usma.edu
Contact: Maj. Jeanne Murtagh
Voice: 937-255-6565 DSN 785-6565
Fax: 937-656-4502 DSN 986-4502
E-mail: jmurtagh@afit.af.mil

Software Engineering Institute Software Engineering Symposium

Dates: Sept. 14-17, 1998
Location: David Lawrence Convention Center, Pittsburgh, Pa.
Sponsor: Software Engineering Institute (SEI)
Contact: SEI Customer Relations
Voice: 412-268-5800
Fax: 412-268-5758
E-mail: customer-relations@sei.cmu.edu

Software Configuration Management Seminars

Dates: Sept. 14-15, 1998
Location: Milwaukee, Wis.
Subject: This course provides an introduction of the disciplines of software configuration management through exercises, demonstrations, and lectures.
Sponsor: Software Configuration Solutions, Inc.
Contact: Voice: 414-938-0442
Fax: 414-938-0443

The Practical Application of Software Configuration Management

Dates: Sept. 16-17, 1998
Location: Milwaukee, Wis.
Subject: This course provides a breakdown of the components of an effective software configuration management environment.
Sponsor: Software Configuration Solutions, Inc.
Contact: Voice: 414-938-0442
Fax: 414-938-0443

Interim Profile Organizational Team Training

Dates: Oct. 20-21, 1998
Location: Pittsburgh, Pa.
Subject: Participants will get the skills to deliver Interim Profile to their organization.
Sponsor: Process Focus Management
Contact: Voice: 916-682-0272
Fax: 916-682-9658

Call for Presentations: International Conference on Practical Software Quality Techniques '98 (PSQT)

Dates: Oct. 5-7, 1998
Location: St. Paul, Minn.
Subject: PSQT focuses only on *practical* techniques. Presentations that reflect real experiences with practical software quality techniques and tools are invited.
Featuring: Watts Humphrey, Boris Beizer, and Robert Binder

Contact: http://www.tcqaa.org/psqt/index.html or http://www.softdim.com/psqt

Earned Value Project Management
A Powerful Tool for Software Projects

Quentin W. Fleming and Joel M. Koppelman
Primavera Systems, Inc.

Earned value can provide any project manager with an early warning tool that sends out a signal from as early as the 15 percent completion point on a project. This signal allows the project manager to forecast the final required funds needed to finish the job within a narrow range of values. If the final forecasted results are unacceptable to management, steps can be taken early to alter the final requirements. The end benefit is that software projects can be completed that contain more final features—if the project's management monitors the true cost performance from the beginning of the project.

Over the last three decades, a proven but yet underutilized project management technique has emerged and taken its place alongside other valuable tools: *earned value*. In its formal application, it has been found to be an effective device to oversee and manage major new systems acquisitions by U.S. government agencies. In a more basic form, earned value can be a useful technique in the management of any project—including, and in particular, software projects.

Earned value requires that the project be fully defined at the outset and then a bottom-up plan be created. This allows measurement to take place during the entire period of performance, from 1 percent to 100 percent of the project's lifecycle. The power in this tool is that it provides accurate and reliable readings of performance from as early as 15 percent into the project. As shown in Figure 1, any project manager can use these performance readings to predict how much it will cost to complete the project within a narrow band of values. If these early warning signals convey unacceptable readings to the project manager, steps can be immediately taken to avoid the undesired results.

This technique is of particular interest to software project managers. No longer must software projects use up all their resources before there is a harsh realization that much of the work has not been completed, forcing features to be dropped to stay within the added budget authorized by management. Earned-value project management can be most helpful to any software project manager who has made a firm commitment to complete all the features within a definitive schedule and for a finite amount of funds.

Introduction to the Earned-Value Concept

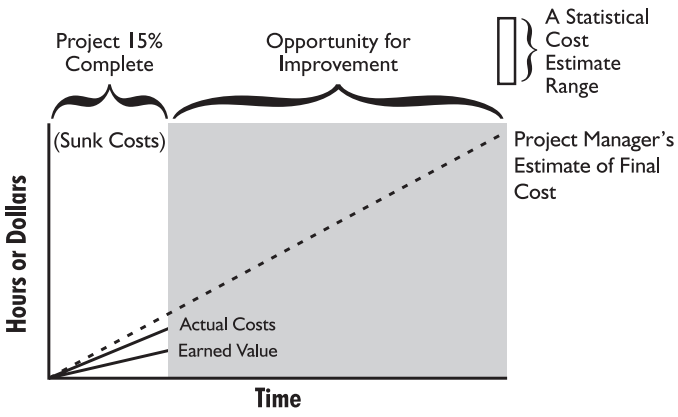
Earned value has been mandated by the U.S. government for decades in an inflexible, formalized manner that has kept many organizations from attempting to use the technique. This mandated, formalized version began in 1967 when the Department of Defense (DoD) issued a directive that imposed 35 Cost/Schedule Control Systems Criteria (C/SCSC) on all private industrial firms that wished to participate in future major government systems in which some type of cost-reimbursable or incentive contract was to be used. Thereafter, any time a new major system would be procured by the U.S. government in which the "risk" of cost growth was retained by the government, these 35 criteria had to be satisfied by the contractor.

The effect of the C/SCSC mandate was to require a formal version of the "earned-value" concept of cost and schedule management on selected major new projects. A certain minimum contract dollar value (in millions) and a minimum program duration (of 12 months or more) had to be present before the criteria were to be applied. Essentially, these earned-value criteria were intended only for major system procurements.

The C/SCSC concept has been consistently applied for over 30 years and has set the standard for major government systems acquisitions. Other government agencies in the United States and in other nations such as Australia, Canada, and Sweden have adopted similar earned-value criteria in the management of their major system acquisitions. A practical body of scientific management knowledge has been developed on the use of the earned-value concept, primarily compiled by the DoD and by the Air Force Institute of Technology (AFIT).

Although some people consider these 35 C/SCSC standards a Utopian ideal for all private firms to emulate, many within private industry have had difficulty employing these

Figure 1. *Cost risks can be managed with an "early warning" signal.*



rigid criteria on all their projects—particularly commercial projects. Their perception is that there are too many nonvalue-added requirements in the formalized C/SCSC for them to be universally employed on all their commercial projects.

Industry's acquired distaste for the C/SCSC implementation of earned value is unfortunate because earned value performance measurement provides a sound project management tool. When properly employed, it can give the project manager an early warning signal that the project is heading for a cost overrun unless immediate steps are taken to change the spending plan. The software world needs something less formal than the full C/SCSC, something that can be scaled downward and precisely tailored to fit broader project management applications. Today, it is likely that more than 99 percent of the projects in the world do *not* employ the earned-value cost management concept. Instead, to monitor costs status, they merely compare their spend plan to their actual costs, and that is unfortunate. There are opportunities to use a simplified form of earned value on any project of any size within the military or commercial sectors.

The Genesis and Evolution of Earned Value

To properly understand the earned-value concept, we must go back in time to the early part of this century and trace the origin of earned value as it came initially from the factory floor.

The Factory Floor in the Early 1900s

The earned-value concept originally came from industrial engineers in factories who for years have employed a three-dimensional approach to assess true "cost-performance" efficiencies. To assess their cost performance, they have been comparing their *earned standards* (the physical factory output) against the *actual costs* incurred. Then they compare their earned standards to the original *planned standards* (the physical work they planned to accomplish) to assess the schedule results. These efforts provided earned value in its most basic form.

Most important, the industrial engineers defined a *cost variance* as the difference between the actual costs spent and the earned standards in the factory. This definition of a cost variance is perhaps the litmus test to determine whether one uses the earned-value concept.

PERT/Cost 1962-1965

The Program Evaluation and Review Technique (PERT) was introduced by the U.S. Navy in 1957 to support the development of its Polaris missile program. PERT attempted to simulate the necessary work to develop the Polaris missile by creating a logic network of dependent sequential events. The initial focus of PERT was on the management of time and on predicting the probability of program success. But before PERT was accepted by program management in industry, the U.S. Air Force came up with an extension of PERT by adding resource estimates to the logic networks. *PERT/Cost* was thus born in 1962, and the initial PERT was thereafter known as *PERT/Time*.

The significance of PERT/Cost, however, was not the technique, but what evolved from it. The earned-value measurement concept was first introduced to the American defense contracting community when the government issued the *DoD and NASA Guide to PERT/Cost* in 1963, which provided a simple definition of earned value. Instead of relating cost plans to cost actuals, which had been the custom, PERT/Cost related the *value* of physical work performed against the cost actuals to determine the utility and benefits from the funds spent. What was *physically accomplished* for what was *spent* was a simple but fundamentally important new concept in program management.

For various reasons, the DoD gave up on the PERT/Cost technique in the mid-1960s but correctly held on to the earned-value concept. When the DoD formally issued the C/SCSC in 1967, the earned-value concept was solidly contained therein.

C/SCSC 1967 to 1996

Since the issuance of the C/SCSC by the DoD, the concept's application has been limited only to contracts in which the government has retained the risks of cost growth, i.e., on cost- or incentive-type contracts and subcontracts. Perhaps the most significant aspect of C/SCSC employment has been the body of scientific knowledge that has been accumulated in its use on major highly technical projects. The DoD has been accumulating data on the use of earned value to assess project performance and has been using the results attained to predict final cost and schedule results with amazing accuracy.

Earned Value Management Systems Criteria 1996 to Present

After years of earned value being imposed on industry by the government as a unilateral mandate, private industry asked for and was allowed to have a say in the wording of the requirements being imposed on them. In 1995, private industry, as represented by the National Security Industrial Association (NSIA), was allowed to assess the utility of the earned-value criteria.

After a year-long study, the NSIA subcommittee came up with its version of the criteria, reworded significantly to be more palatable to the project management community. The industry standard was called the Earned Value Management System (EVMS) and the number of criteria was reduced from 35 to 32. This major development was endorsed by the DoD in December 1996.

However encouraging these recent advancements may be, going from 35 to 32 criteria still leaves the earned-value concept with far too many nonvalue-added requirements. We believe the earned-value concept will never be universally accepted by project managers in its current form, embedded as a part of the 32 formal EVMS criteria. There are too many rules and terms one must master to employ this approach. Instead, what is needed is a return to the simple concept that originally came from the industrial factory floors. The industrial engineers did not use checklists and interpretations to employ their concept;

rather, they used common sense to determine what was needed and what did or did not work.

Listed below are 10 earned-value "musts" that, when followed, capture the critical essence of the earned-value concept and enhance the management of all projects, large and small, from any industry.

Ten Musts to Implement Earned Value on All Projects

Define Work Scope

You must define 100 percent of the project's work scope using a work breakdown structure (WBS). Perhaps the most critical and most challenging requisite to employing earned value is to define the project's total work scope. This is a difficult task for any project, and particularly so for software projects. Yet, if you do not define what constitutes 100 percent of the assumed work, how can you measure the project's performance in a definitive way? Without a 100 percent reference point, how can anyone ascertain whether you have completed 10 percent, 20 percent, or 25 percent of a job?

Realistically, no one can define a new job with absolute precision, but you must make some intelligent assumptions about a new project to quantify the work with sufficient confidence that the defined effort can be planned, scheduled, and estimated with some degree of certainty. Anything less, and management must commit to a job by authorizing a "blank check" for the project.

How does one define a job when specific details are often lacking? There are no absolute answers, but one of the most useful of all tools available to any project manager is the WBS. The WBS is to the project manager what the organization chart is to the executive—it allows the project manager to define a new endeavor by laying out all the assumed work, then decomposing each task into measurable work packages. Once the WBS is assumed to constitute a reasonable portrayal of the new project, it can be used to take the next steps in the project planning process, including the make-or-buy analysis, risk assessment, planning, scheduling, estimating, and authorization to proceed.

Create an Integrated Bottom-Up Plan

You must combine critical processes, including defined work scope, schedule, and estimated resources, into an integrated bottom-up plan of detailed measurement cells called Control Account Plans (CAPs). Earned value project management is implemented within detailed CAPs, which therefore constitute formal bottom-up project planning. The individual CAPs represent the integration of all critical processes such as work scope, planning, scheduling, estimating, and authorization.

The performance measurement will take place within the detailed CAPs, and the total project's performance is the summation of what was reflected in the detailed CAPs. In essence, each project CAP is a subproject of the total project that is managed, measured, and controlled by a CAP manager.

Formally Schedule CAPs

Each of the defined CAPs must be planned and scheduled with a formal scheduling system. This is perhaps the single most critical tool required to implement earned value. The project's scheduling system will portray the approved work scope, which is carefully placed into a specific timeframe for performance. In earned-value vernacular, this scheduled work will constitute the project's *planned value*. As performance takes place on the project, the portion of the planned value that is physically accomplished becomes the earned value. Both the planned value and the resulting earned value must use the same metrics to measure their performance.

The project's scheduling system is, therefore, critical to the employment of earned value because it is the vehicle to represent the project's scope, planned value, and earned-value measurement. The *project master schedule* is vital to the project because it constitutes the project manager's specified planned value for everyone to follow.

Assign Each CAP to an Executive for Performance

Each of the defined CAPs must be assigned to a permanent functional executive for performance. This assignment effectively commits the executive to oversee the performance of each CAP. Projects are by their nature transient within any firm's permanent organizational structure—they are authorized, implemented, and performed, then eventually go out of existence. Many (perhaps most) of those who manage the detailed performance that takes place within the CAPs will not carry the formal title of "manager" within the firm's permanent organizational structure; rather, many or most of these CAP managers are functional employees temporarily assigned and matrixed into the project by one of the permanent functional organizations. To secure a firm commitment from the functional executives who have the authority and resources to make the plan happen, it is wise to have each of the defined project CAPs essentially adopted by a senior function person with a title such as vice president, director, or manager.

Establish a Baseline that Summarizes CAPs

A total project performance measurement baseline must be established, which represents the summation of the detailed CAPs. The next required step is to form a total baseline against which project performance may be measured. Such baselines must include all defined CAPs plus any management (contingency) reserves that may be held by the project manager. If management reserves are not given to the project manager but are instead controlled by a senior management committee, they should be excluded from the project performance baseline.

On a commercial-type contract, the baseline may include such things as indirect costs—and even profit or fee—to match the total authorized project funds. Internal projects will typically not contain indirect costs, profits, or management reserves. Most internal project

baselines will be the sum of the defined CAPs.

Measure Performance Against Schedule

Periodically, you must measure the project's schedule performance against its planned master project schedule. The formally issued and controlled project master schedule constitutes the project's planned scope. Each task described on the project master schedule can be loaded with estimated resources, such as hours or dollars, which are embedded within the authorized CAPs. As performance takes place within the CAPs, you can quantify the relationship between the value of the work scheduled as compared to the value of the work accomplished. The difference between the work scheduled and work accomplished constitutes the *schedule variance* in earned value.

A negative schedule variance means that the value of the work accomplished does not match the value of the work scheduled, i.e., the project is falling behind in its scheduled work. Each behind-schedule task can be assessed regarding its criticality to the project. If the late task is on the critical path, or if the task carries a high risk to the project, efforts can be made to get the late task back on schedule. Conversely, if a task has positive variance or is not considered a high risk to the project, added resources should not be spent to accelerate its performance.

Measure Cost Efficiency Against the Costs Incurred

You must periodically measure the project's cost performance efficiency rate, which represents the relationship between the project's earned value performed and the costs incurred to achieve the earned value. The single most important benefit of employing earned value is the cost efficiency readings it provides. The difference between the value of work performed and the costs incurred to accomplish the work provides the cost-efficiency factor. If you are spending more on the project than it receives in value, this reflects an overrun condition. Absolute overruns have been found to be nonrecoverable. Overruns expressed as a percentage value have been found to deteriorate unless the project takes aggressive actions to mitigate the condition.

Perhaps of greatest benefit, the cost efficiency rate has been found to be usably stable from the 15 percent point of a project completion and progressively more stable as it goes from the 20 percent to 30 percent to 40 percent completion point. Therefore, the cost efficiency factor is an important metric for any project manager or enterprise executive to monitor.

Forecast Final Costs Based on Performance

Periodically, you must forecast the project's final cost requirements based on its performance against the plan. One of the more beneficial aspects of the earned-value concept is its ability to independently forecast the total required funds at the end of a project, commonly called the "estimate at completion." Based on project performance against the plan, a project manager can accurately estimate the total funds required to finish the job within a finite range of values.

These statistical estimates are something like a grass-roots sanity check against estimates based more on wishful thinking because they provide a more realistic estimate of the values needed to finish the job—unless someone has a preconceived notion of what that value should be. As reflected in Figure 1, if the earned-value statistical estimates are greater than the "official" project estimates to complete the project, someone in a senior management position should reconcile these professional differences of opinion.

Manage Remaining Work

You must continuously manage the project's remaining work. The results achieved to date on a project, good or bad, are in effect "sunk costs"—gone forever. Thus, any improvements in performance must come from future work—tasks ahead of the latest status date. Earned value allows the project manager to accurately measure the cost and schedule performance achieved to date. If the results thus far are less than desired, the project manager can exert a more aggressive posture on all future work. Earned value, because it allows the project to accurately quantify the value of its work achieved, allows the project manager to also quantify the value of the work ahead to stay within the objectives set by management.

Manage Baseline Changes

You must continuously maintain the project's baseline by managing all changes to the baseline. The project performance measurement baseline you put in place at the start of the project is only as good as your management of all proposed changes to the baseline during the duration of the project. Any performance baseline quickly becomes invalid if you fail to incorporate changes into the approved baseline either by the addition to or elimination of added work scope.

All new changes of project work must be addressed either by the approval or rejection of changes. For the initial baseline to remain valid, every change must be closely managed. Maintaining a baseline is as challenging as the initial definition of the project scope at the start of the project.

Conclusion

The earned value project management concept as a part of the more formal C/SCSC or EVMS has been demonstrated to be an effective technique in the management of major projects. Unfortunately, most of the experience with the concept has been restricted to those applications where the U.S. government has imposed the technique on major new systems acquisitions for which it retains the risk of cost growth.

However, the best opportunities for earned-value employment may well lie in the management of thousands of smaller projects that are being directed by people who may well be unaware of earned value. We believe the concept should be considered any time the risk of cost growth resides with a project manager, any time a lump sum or fixed price contract is used, and on all in-

house funded developmental projects where a firm commitment is made to management. It should be considered any time a project manager could benefit from receiving an early warning cost signal in time to alter the ultimate direction of a project. Software projects can especially benefit from the employment of a simple earned-value approach. ◆

About the Authors

Quentin W. Fleming, senior staff consultant to Primavera Systems, Inc., has over 30 years industrial project management experience. He held various management assignments with the Northrop Corporation from 1968 until 1991, served on an earned-value corporate review team, and wrote the corporate policy directive on scheduling.

He is president of the Orange County Project Management Institute (PMI) chapter and developed and taught four PMI Project Management Professional tutorial courses covering scope, cost, time, and procurement management. He has a bachelor's and a master's degree in management and is the author of seven published textbooks including *Earned Value Project Management*, which he co-wrote with Joel M. Koppelman.

E-mail: QuentinF@Primavera.com

Joel M. Koppelman is president of Primavera Systems, which provides a family of project management software products. Before co-founding Primavera in 1983, he spent over 12 years planning, designing, and managing major capital projects in the transportation industry, including duties as vice president and chief financial officer for Transportation and Distribution Associates, Inc. Before that, he was affiliated with the management consulting firm of Booz Allen Hamilton, Inc.

Koppelman is a registered professional engineer with a bachelor's degree in civil engineering from Drexel University and a master's of business administration degree from the Wharton School of the University of Pennsylvania. He is a frequent speaker at universities and for international management organizations.

E-mail: JKoppel@Primavera.com

Year 2000 Readiness Checklists

Watts S. Humphrey
Software Engineering Institute, Carnegie Mellon University

The checklists in this article are designed to help organizations determine their readiness for the year 2000 (Y2K) date change. The lists are brief but include the items many experienced professionals have concluded are necessary for adequate Y2K preparation. They will help managers quickly assess their status and determine where they are exposed. This article also includes explanations of the various items and some general comments.

Recently, I was invited to a meeting in Washington D.C. to review a proposed quality standard for the year 2000. Although the meeting was focused on testing, the discussion reinforced the multifaceted nature of the Y2K problem. Since few people have seen a problem like this, we need to characterize it in a way that clearly identifies the actions that must be taken, and a checklist could be what is needed. It would describe the actions required and enable managers to quickly see the work they need to get under way.

After drafting the Y2K readiness checklist, I had several knowledgeable people review it. Although no brief checklist can be complete, this one covers the points the reviewers and I felt were most important. Organizations may identify additional important areas, but they should not substantially expand the checklist because that would make it harder to use and less effective.

This checklist is designed to help you judge the readiness of your organization for the Y2K transition. In using this checklist, you should consider several points.

- The Y2K transition problem will be much like those you have experienced in installing and converting to a new system or application program version, only this time, you will be installing and converting to an updated version of every application and system simultaneously.
- If you experience Y2K problems with any commercially supplied software package, other users will likely have similar problems at the same time. Thus, the vendors' help desks and support services will probably be swamped with calls and unavailable for extended periods.
- Unless you have contracted for dedicated support services, you must be prepared to be self-sufficient. If you have a support contract, you need to ensure the suppliers of such services have resources dedicated to your needs.
- The Y2K cutover will most likely result in multiple application and system crashes and other disasters. It is therefore essential that you maintain complete backups of all application and system data and programs. Note, however, that with the Y2K problem, traditional backup practices will not work. Usually, when backing up, one returns the system to a prior configuration that worked. In this case, at least until after the cutover period, there will be no prior configuration that is known to work.
- Be careful about vendor selection because there will likely be unscrupulous providers of Y2K services. When organizations do not have a competent technical staff, dishonest operators could pretend to do the required work, then disappear before Jan. 1, 2000. Any organization that wants substantial payments upfront should be avoided like the plague. Although scams are nothing new, Y2K is an extremely attractive opportunity; any organization that is victimized could well be out of business before it can recover damages.
- View all Y2K tools, services, guidelines, and checklists (including this one) with a high degree of skepticism. Remember, none of them have been tested in practice and shown to work. You therefore must examine all such offerings and satisfy yourself that they are credible and reliable.
- Consider each specific item in this checklist and satisfy yourself that it is necessary for your organization. With the exception of the crisis response, zero-hour testing, and emergency backup, all other items except one may or may not be essential, depending on your situation. Although you may not need these essential items, if you do (and most organizations will) you cannot build them on short notice.
- The one capability exception that every organization should put in place as soon as possible is a configuration management system. If you do not have this capability, you will most likely have problems that could be severe and unrecoverable. This must not be viewed as something to do later when you have a chance. Put a configuration management system in place now.

The Y2K Readiness Checklists

The following checklists are designed to help you quickly assess the readiness of your organization for the Y2K transition. Complete the organization checklist first (Figure 1), then fill out the application checklist for every active application (Figure 2).

This article is based on "What Does Y2K Mean to You," Object Magazine Online, April 1998 (http://www.sigs.com/omo).

Completing the Checklist

Have the most knowledgeable engineers and managers complete the checklists. Brief descriptions of the checklist terms follow the checklists. In completing the checklist, the columns to the right refer to the status of the application or the overall organization. If all the required work has been done, check the "done" column. Similarly, if the work is not yet done but the project is staffed and under way, check the "under way" column. Later, when the application checklists are completed, you can enter percentage values for the percent of applications that have been completed for each checklist category.

Figure 1. *The Y2K organization readiness checklist.*

*	Organization Units (laboratory, plant, etc.):	Done	Under Way	Plan	No Plan
	Should be done by Jan. 1, 1998.				
1	Adequate 1998 budget and staff in place.				
2	Configuration management system in place.				
3	Applications inventoried.				
4	Applications assessed (with checklist).				
5	Application source code under control.				
6	Application priorities determined.				
7	Y2K tools available.				
8	Date change standards defined and tested.				
9	Database correction inventoried.				
10	Service and support facilities surveyed for Y2K.				
11	Critical applications staffed and in development.				
	Should be done by Jan. 1, 1999.				
12	Adequate 1999 budget and staff in place.				
13	Applications updated and in test—*critical*.				
14	All applications staffed and in development.				
15	Database corrections staffed and in development.				
16	Service and support corrections under way.				
17	Hot-line group funded and staffing identified.				
	Should be done by July 1, 1999.				
18	Backup procedures defined.				
19	Applications updated and in test—*key*.				
20	Emergency procedures defined and tested.				
21	Customer and supplier testing under way.				
22	Zero-hour procedures defined.				
23	Hot-line groups staffed and supporting testing.				
24	Database corrections in test.				
	Should be done by Sept. 1, 1999.				
25	Emergency procedures training in place.				
26	Applications updated and in test—*all*.				
27	Zero-hour procedures tested.				
28	Backup procedures tested.				
29	Service and support tests completed.				
	Should be done by Dec. 1, 1999.				
30	All application and database testing complete.				
31	Adequate Y2K budget and staff in place.				
32	Customer and supplier testing completed.				
33	Emergency procedures training complete.				
34	Backup procedures in full operation.				
35	Zero-hour testing staffed for Jan. 1, 2000.				
36	Hot lines fully staffed and rehearsing procedures.				
	Should be done by Jan. 1, 2000.				
37	Emergency procedures rehearsals completed.				
38	Zero-hour testing under way for Jan. 1, 2000.				
39	Zero-hour testing staffed for Feb. 19, 2000.				

The Y2K Application Checklist

Complete the application checklist for every active application in the organization.

Checklist Definitions – Checklist Columns

The four columns to the right of the checklist are for status information. In most cases, this work is either done or not done. Generally, it is desirable to either check the item or enter a date when it will be done. After an initial assessment, and after the work is under way, it is helpful to enter a percentage figure, as with "Applications assessed (with checklist)." Here you would enter the percentage of the applications assessed. Note, however, that the columns should add to 100 percent when using percentage values for a checklist row.

It also is important to only count completed work. For example, if you have 10 items and three of them are finished, that would be 30 percent complete. When six of 10 items are each half done, however, you would have zero percent done. Do not take credit for partially completed work because partial status is generally hard to estimate and can be misleading.

The status levels are defined as follows:

Done – This column is checked when the listed work has been completed.

Under Way – This column is for work that is staffed and under way. In those cases where the work is only 50 percent staffed, for example, it would be more informative to enter a 50 percent in this column. Note that this column does not give any indication of whether the work is likely to be completed on schedule.

Plan – This column is for those cases where the work is planned but it is not yet staffed or the staff may have been identified but the work has not yet started.

No Plan – This column is for those areas where the work has not yet been planned

or it has been planned but no staff has yet been identified to do the work.

***** – Where any tasks are late or need special attention, mark them with an asterisk in the "*****" column.

Application Priorities

The definition of critical applications is a matter of judgment and must usually be settled by senior management. It is therefore suggested that a comprehensive listing of all applications be reviewed with management together with a list of those applications that are deemed critical and why. This is the action called for under "Application priorities determined." These priorities should specify which programs are critical, key, active, and inactive and also which will need emergency backup procedures, hot-line support, or zero-hour testing.

The application priorities are as follows:

Critical – These are the applications on which the organization's business depends. They can be found by identifying the work that would have to be done manually if all computers were shut down for weeks or months. Without the critical applications, the business could not function.

Key – The key applications are those the business needs but are not a matter of organizational survival. Although they are needed, their unavailability for periods of weeks and even a few months would not be fatal; that is, either the application work can be deferred or manual back-up procedures will be devised to handle the needs in the interim.

Active – The active applications are those actively in use other than critical or key applications. These applications may only be run once a year or occasionally on demand. Although they are needed, their repair can be deferred until shortly before the application is needed. Note, however, that some applications may only be used once a year for tax purposes. If that one-time use is in January, however, this could become a critical application.

Inactive – These are all the applications that are no longer in active use. In most cases, these would not warrant a Y2K repair effort.

Readiness Dates

The dates given in the checklist are selected based on overall judgment of the amount of work to be done in a small- to medium-sized organization that has a competent software staff on hand. These are the latest advisable dates; organizations should strive to get this work done earlier if possible. Also, if the organization is extremely large or if it does not have a reasonably large and competent staff, these dates should be even earlier.

For large organizations, earlier dates are needed because of the huge volume of work. Smaller organizations without a substantial information systems staff will need to hire one or more suppliers of Y2K services. It takes time to identify and obtain these services, so these organizations should work to meet the earliest possible dates.

Alphabetical Glossary

Following is an alphabetical listing of definitions of many of the items on the readiness checklists. The topic headings are the same as in the checklists, and the numbers refer to the numbers in the asterisk column.

(1, 12, 31) Adequate (year) budget and staff in place – This should be management's top priority. Unless the

Figure 2. *The Y2K application checklist.*

	Application Name:				
	Application Priority:				
	Application Support Needs			Yes	No
	Emergency procedures required.				
	Hot-line capability required.				
	Zero-hour testing required.				
*		Done	Under Way	Plan	No Plan
	Should be done by Jan. 1, 1998.				
50	Application priority determined.				
51	Source code available.				
52	Source code checked against object code in use.				
53	Critical applications: staffed and in development.				
54	Applications under configuration management.				
55	Database correction inventoried.				
	Should be done by Jan. 1, 1999.				
56	Applications updated and in test—*critical*.				
57	All active applications: staffed and in development.				
58	Database corrections staffed and under way.				
59	Hot-line groups funded and staffing identified.				
	Should be done by July 1, 1999.				
60	Emergency procedures defined and tested.				
61	Applications updated and in test—*key*.				
62	Critical applications tested with updated database.				
63	Hot line partially staffed and supporting testing.				
	Should be done by Sept. 1, 1999.				
64	Emergency procedures training in place.				
65	All applications updated and in test.				
	Should be done by Dec. 1, 1999.				
66	All application testing complete.				
67	Emergency procedures training complete.				
68	Hot-line fully staffed and rehearsing procedures.				
	Should be done by Jan. 1, 2000.				
69	Emergency procedures rehearsals completed.				
70	Zero-hour testing under way.				

work is staffed in time, there is no way to finish in time. The adequacy of the funding and staffing should be based on an assessment of a plan to do the work and an estimate of the resources required.

(4) Applications assessed (with checklist) – This refers to applications being assessed with the application checklist.

(3) Applications inventoried – Every application in use must be identified. This requires naming the application and where and when it runs. The inventory also should list available source code, manuals, procedures, and guidelines about the application, who uses it, and when. This information is needed to set priorities.

(6, 50) Application priority determined – Management must decide which applications to fix, which to replace, and which to handle with a hot-line capability. This sets the priorities for every application and guides the allocation of development work. If these decisions are not made early in the Y2K program, important programs will likely be overlooked while less critical applications are being fixed. Management must set priorities at the earliest possible point: which applications are critical, key, active, and inactive.

(5) Application source code under control – see "Configuration management" and "Source code available."

(54) Applications under configuration management – see "Configuration management."

(13, 19, 26, 56, 61, 65) Applications updated and in test – once corrected, the applications must be tested with the corrected databases. It is important that this testing cover the applications' functions as well as all the code and database changes.

(18, 28, 34) Backup procedures – The Y2K cutover will most likely result in multiple application and system crashes and other disasters. It is essential that complete backups be maintained of all application and system data and programs and that these backups be updated frequently. All old backups must also be retained since files can be unknowingly corrupted and not discovered until much later. Good practice dictates that backups be taken as early as possible, even before Y2K work starts. Note, however, that with Y2K, traditional backup practices will not work. Usually, when backing up, one returns the system to a prior configuration that worked. With Y2K, at least until after the cutover period, there will be no prior configuration that is known to work.

(2, 54) Configuration management – The configuration management system maintains physical and electronic control of the organization's programs and data. Applications that have been in use for many years often have not been changed for much of that time. Unless the organization has an established configuration management system, the source code could have been lost. The configuration management system is also needed to ensure that the changed programs are properly updated in test and that only tested programs are put into use. Without an effective configuration management system, organizations are likely to lose programs, misapply fixes, or use the wrong tests and test data. All this wastes time, which is the one thing you cannot recover.

(21, 32) Customer and supplier testing under way – Many businesses have critical dependencies on their customers or suppliers. Where these relationships involve data processing interactions, there will likely be Y2K problems. The fixes to these problems must be tested in advance.

(9, 15, 24, 55, 58) Database corrections – Depending on the Y2K change strategy, many database changes may have to be made. Also, during the transition, there are many ways that databases can be corrupted. Until programs have been corrected, for example, many date calculations will put incorrect dates in the database. After the programs are corrected, subsequent dates will be correctly calculated. The application, however, will not go back and search for the incorrect entries in the old data. This must be done by hand or with special tools, if any can be found. This issue is further complicated by the phased cutover of multiple applications. The corrected databases must then be tested with the corrected applications.

(8) Date change standards defined and tested – The Y2K date conversion formulas are not complex, but they are not trivial. It is essential that the engineering change teams know precisely how to handle date calculations.

(20, 25, 33, 37, 60, 64, 67, 69) Emergency procedures defined – With large systems and large volumes of changes, there will be many defects. Thus, even the most critical applications will likely be unavailable for periods. The organization must be prepared for this eventuality and have a capability in place to handle any problems. The emergency procedures define how an application is handled under these conditions. Manual procedures should be in place and tested for all critical and selected key applications. Because large numbers of people will need to know how to quickly respond in an emergency, training programs and procedure rehearsals will generally be needed.

(17, 23, 36, 59, 63, 68) Hot lines – The hot-line support group handles the crisis calls when applications fail during and after the Y2K cutover. Since application failures can occur early for applications with advanced dates (like credit card expirations), the hot-line groups may have to be staffed much earlier, depending on application needs. It also is important to staff these groups early to give them experience with the applications they will handle. The best way to do this is to have them in place handling Y2K testing problems and fixes.

A hot-line support capability will be needed even for those applications that have been completely repaired and tested, because between 1 percent to 20 percent or more of all the Y2K fixes will

likely have problems, even after testing. With a fix quality program in place, the 1 percent number is achievable. If not, 20 percent or more is likely, depending on staff experience, program complexity, and the degree of testing.

Generally, you will need application-knowledgeable people to staff the hot lines. They provide telephone assistance to system and application users, internal or external. Their job is to help the application users when they run into problems.

(10, 16, 29) Service and support – Many facilities such as elevators, security systems, power distribution, telephone systems, and air conditioning could have date dependencies. Although these items may not have problems, they could. Each one should be tested or checked with the manufacturer for Y2K compliance and warranty coverage.

(51) Source code available – If the source code for any active program is not available and the program has date dependencies, the application must be replaced and tested. Replacement may be expensive and take time, but without the original author, programs can rarely be fixed without the source code.

(52) Source code checked against object code in use – In older systems, applications were occasionally patched to avoid the time required to recompile and rebuild. When this has been done, the source code will not represent the program that is being used. When the developers update the source code for the Y2K fixes, compiling and installing it will erase all these prior object corrections. The result will be the simultaneous re-emergence of all the problems the original object patches were designed to fix. These problems will not wait until the year 2000; they will happen starting now. To resolve such problems, these patches must be identified and added to the Y2K workload. These patches can be found by compiling a new object program from the source and making a bit-for-bit comparison with a copy of the object program in use.

(7) Y2K tools available – This box should only be checked after the tools have been evaluated, obtained, and tested in practice. Until they are, it is likely that many of the tools will be found less effective or harder to use than promised.

(22, 27, 35, 38, 70) Zero hour – The zero hour is midnight Friday, Dec. 31, 1999. Over the long weekend of Jan. 1, 2000, the large and complex Y2K system change must be tested live for the first time. The zero-hour procedures define how the time between Thursday, Dec. 30, 1999 and Tuesday, Jan. 4, 2000 is to be used. During this period, many groups should make test application runs both in-house, with suppliers, and with customers. This weekend is also when the hot-line support groups must move up to full capacity. The zero-hour procedures are used to phase repaired applications into service and recover from any problems found. Although this cutover should be started as early as possible, special recovery resources and procedures must be available and tested during zero-hour testing. Plan to maintain the zero-hour testing capability until all the critical applications are cut over and work properly. This will likely take at least a month and could take much longer. Some organizations plan to maintain this special testing and support capability for at least three months.

(39) Zero-hour testing under way for Feb. 29, 2000 – Normally, there is a leap year every four years. The exception is every 100 years when there is not a leap year. This too has an exception every 400 years when there is a leap year. Thus, 2000 is a leap year, and there will be a Feb. 29, 2000. It is important that the date change algorithms be clearly defined and disseminated so everybody working on this problem understands them. Since these date algorithms will first be tested Feb. 29, 2000, a zero-hour testing plan is needed.

Summary

Any organization that does not have an active Y2K program under way had best get started immediately. The date when the work will be completed is principally determined by when the work starts. If you are still studying, stop and get to work. Make a plan at the same time, but get to work! ◆

About the Author

Watts S. Humphrey is a fellow at the Software Engineering Institute (SEI) of Carnegie Mellon University, which he joined in 1986. At the SEI, he established the Process Program, led initial development of the Capability Maturity Model, introduced the concepts of Software Process Assessment and Software Capability Evaluation, and most recently, the Personal Software Process and Team Software Process. Prior to joining the SEI, he spent 27 years with IBM in various technical executive positions, including management of all IBM commercial software development and director of programming quality and process. He has master's degrees in physics from the Illinois Institute of Technology and in business administration from the University of Chicago. He is the 1993 recipient of the American Institute of Aeronautics and Astronautics Software Engineering Award. His most recent books include *Managing the Software Process* (1989), *A Discipline for Software Engineering* (1995), *Managing Technical People* (1996), and *Introduction to the Personal Software Process* (1997).

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213
Voice: 412-268-6379
E-mail: watts@sei.cmu.edu

Software Surprise

The Three Invisible Problems of Weapons System Software Development

Lt. Col. Lionel D. Alford Jr.
U.S. Air Force

This article describes how software-induced workload, software system complexity, and software systems cost may cause rarely identified but long-lasting adverse effects to a program. If you cannot find these three problems in your software development program, you may not realize what hit you until it is too late.

Software in developmental systems causes three critical challenges for the manager of a system program office (SPO), the testers, and ultimately the customers: software-induced workload, software system complexity, and software systems cost. For each of these challenges, you could insert the word "integrated" in place of "software"; the result is the same. Even though these three problems have an enormous effect on the overall system, they are given little attention because SPOs rarely realize they exist. All three of these problems are "program invisible"—they are rarely tested or given any thought until after they have become a serious difficulty for the SPO. The problem is that these software and integration problems are some of the foremost reasons for customer dissatisfaction and increased systems cost.

Software-Induced Workload

SPOs attempt to reduce software-induced workload by adding software to the system. Current hardware and the missions this hardware supports are extremely complex. Software is primarily used to integrate and consolidate complex systems so the equipment operators can accomplish the mission with decreased workload and increased mission effectiveness. However, no one has discovered a way to measure workload. All the measures we currently have for workload are qualitative and not quantitative.

In the past, engineers tried to use quantitative measures such as altitude and airspeed capture to measure workload; unfortunately, these measures have nothing to do with workload. Take the example of a test pilot who is required to use digital instruments to keep an aircraft within 10 feet above or below a target altitude. According to conventional engineering measures, the workload should not be great because the event falls within the realm of possibility; nevertheless, the workload is extremely high—the pilot has to constantly work the controls and interpret instruments. Even a test pilot cannot accomplish this task for long. After a series of engineering analyses, tape altimeters were installed on the C-5, C-141, F-111, and FB-111 aircraft. Aviators who have flown these aircraft will testify to their "low" workload after they have become proficient in the systems; however, controlled analytical tests with other aviators using standard instruments always show that their perceptions are inaccurate.

Because there is no usable measure for workload, when we try to measure workload, data from such analyses are always suspect: The sample size is rarely large, the statistical confidence is low, and there is no method to quantitatively measure the workload. Since we use these analyses when evaluating whether we want to reduce the number of crew members in the cockpit, for instance, it is not a decision based on analysis and test; it is a hope based on politics and the cost of the additional crew members. The best examples of this are the MC-130H aircraft and the current U.S. Air Force glass cockpits and heads-up displays (HUDs). On the one hand, the MC-130H is one of the best missionized aircraft in the world. The pilot puts the cue on the dot and can fly any terrain by following profile programmed by the navigator and the aircraft system. On the other hand, it is a poor instrument aircraft. The tape digital displays make it extremely difficult to fly. In like fashion, the glass cockpits and HUDs of Air Force aircraft are based on similar tape displays. These displays work well for civil aircraft, which are flown from takeoff to touchdown on the autopilot, but they are "workload sinks" for military tactical flight. This workload problem will continue to be an obstacle until a method to quantitatively measure workload is discovered.

Fortunately, there is research toward this end, but a majority of fielded and future systems have been or are being designed without regard to the workload involved. A final example is radio frequency changes in aircraft that use digital integrated radio systems. It is simple to change a frequency using the old analogue dial paradigm—the pilot inputs the frequency by turning a dial on the console. But in a software display, the pilot must first find the page for frequency entry, then select the proper place for the entry, and finally, input the digits from a touch pad. This is at least 10 times greater workload than the analogue dialing system, yet it is the new paradigm. If you multiply the workload in this example by the number of system inputs the pilot must make to accomplish any mission, it will demonstrate only a small fraction of the magnitude of problems associated with workload. It is enough to say that software and integrated systems generally have significantly increased workload without a proportional increase in mission effectiveness.

Software Complexity

The second great hidden problem in software development is software com-

plexity. Because software is so intrusive—that is, it affects many systems—it has become impossible to fully test even the safety-related effects of the software. When a new software build is installed in an aircraft, unknowns are rampant, and the "bugs" are rarely fully discovered during flight test. Some problems lie dormant until the systems are well deployed. One example was an operational flight program (OFP) release on the MC-130H. This release was supposed to affect only the terrain-following (TF) system of the aircraft. The aircraft was released for flight under the assumption that it would operate properly as long as the TF system was not engaged. In the middle of a training flight, during an engine-out approach, the crew noticed that the "ball" (primary flight coordination instrument) was indicating sideslip in the opposite direction. Because the TF system was an integral part of the OFP, a change to the TF system software resulted in an erroneous reading in another part of the system. If this OFP had made it into the fleet, or an experienced test crew had not been flying the aircraft, it is likely there would have been a smoking hole where a multimillion-dollar aircraft once had been. This is an extreme example, but there have been hundreds of others in and out of flight test.

Software and integrated systems increase risk proportional to the increasing code and increasing integration complexity. In the C-21 aircraft (Lear 35), a pulled or popped oil pressure circuit would cause the engine control settings to indicate fire on an engine. An operational crew discovered this problem when they got two fire lights, one on each engine. They had to shut down a good engine and land short of their destination. They were lucky to realize there was a problem with the indicating system before they shut down both engines. The circuit breaker had popped due to a faulty circuit problem, and a sneak circuit caused the fire warning in the indicating system. A $10 piece of equipment gave the software false information, and the crew and passengers were placed at risk because testing had not been done with the oil pressure circuits pulled. This defect has been fixed since the incident, but who knows how many similar problems wait to be found? Software and integration complexity increase risk.

Software Systems Cost

The third problem is related to the first two. Software always requires future improvements and rewrites. Complex software invariably comes with bugs that are never entirely discovered. Modifications and fixes add more bugs, which results in future modifications and fixes. Rarely are software systems provided with sufficient lifecycle funding for these processes.

Software has become so intrusive that the simplest components on many aircraft incorporate some software. In fact, such things as the clocks, circuit breakers, and pressurization systems in most modern aircraft incorporate or are dependent on software for correct indication and operation. Most aircraft are now to some degree fly-by-wire and engine control-by-wire. This trend toward software-driven controls and systems shows no sign of change or reversal. Therefore, funding must be provided for any software system until the decommission of the system—a given that has not been acknowledged by most services and program offices. For example, there are numerous electronic warfare systems that are not adequately funded for software changes but are nevertheless going through major changes. This has resulted in serious program problems such as multiple OFPs in multiple versions being deployed by more than one agency. The resulting costs are much more than they would have been if software changes had been planned for the life of the system. The examples of the MC-130H and the C-21 resulted in unplanned cost increases that could have radically affected the safety of the aircraft if the funding had not been made available.

Conclusion

The lessons to learn from these three invisible software and integration problems are simple—their solutions are not. First, try to evaluate workload when developing a system. Attempt to use nonintegrated systems when possible, especially when workload studies indicate a problem. The Department of Defense must fund research and development to discover effective quantitative workload measures. Second, plan and test for as much as possible, and be ready—during all program phases—for software problems to rear their ugly heads. Do not be content with minimal software testing even when risk is low. Finally, fund software for the life of the system.

These three issues are critical, rarely visible problems. They should be primary considerations during all SPO phases. They may be invisible now, but unless tamed, they will drive your program and the capability of your weapons system. ◆

About the Author

Lt. Col. Lionel D. Alford Jr. is the chief of the Special Operations Forces Test and Evaluation Division, Wright-Patterson Air Force Base, Ohio. He is an Air Force experimental test pilot with over 3,600 hours in more than 40 different type aircraft and is a member of the Society of Experimental Test Pilots. Alford has served as the chief of the Testing Commercial Aircraft for Military Acquisition Office at Edwards Air Force Base, Calif., holds an Airline Transport Pilot license, and was the chief test pilot for a number of Air Force acquisitions. He is a graduate of the Defense Systems Management College Advanced Program Management Course 98-1. He has a master's degree in mechanical engineering from Boston University and a bachelor's degree in chemistry from Pacific Lutheran University.

ASC/LUQ
2275 D Street, Room 142
Wright-Patterson AFB, OH 45433
Voice: 937-255-9311
Fax: 937-255-0995
E-mail: Pilotlion@aol.com

CROSSTALK

Sponsor	***Lt. Col. Joe Jarzombek*** 801-777-2435 DSN 777-2435 jarzombj@software.hill.af.mil
Publisher	***Reuel S. Alder*** 801-777-2550 DSN 777-2550 publisher@stsc1.hill.af.mil
Managing Editor	***Forrest Brown*** 801-777-9239 DSN 777-9239 managing_editor@stsc1.hill.af.mil
Senior Editor	***Sandi Gaskin*** 801-777-9722 DSN 777-9722 senior_editor@stsc1.hill.af.mil
Graphics and Design	***Kent Hepworth*** 801-775-5555 ext. 3027 graphics@stsc1.hill.af.mil
Associate Editor	***Lorin J. May*** 801-775-5555 ext. 3026 backtalk@stsc1.hill.af.mil
Editorial Assistant	***Bonnie May*** 801-775-5555 ext. 3022 customer_service@stsc1.hill.af.mil
Features Coordinator	***Heather Winward*** 801-775-5555 ext. 3023 features@stsc1.hill.af.mil
Customer Service	801-777-8045 custserv@software.hill.af.mil
Fax	801-777-8069 DSN 777-8069
STSC On-Line	http://www.stsc.hill.af.mil
CROSSTALK On-Line	http://www.stsc.hill.af.mil/Crosstalk/crostalk.html
ESIP On-Line	http://www.esip.hill.af.mil

Subscriptions: Send correspondence concerning subscriptions and changes of address to the following address:

Ogden ALC/TISE
7278 Fourth Street
Hill AFB, UT 84056-5205

E-mail: custserv@software.hill.af.mil
Voice: 801-777-8045 DSN 777-8045
Fax: 801-777-8069 DSN 777-8069

Editorial Matters: Correspondence concerning *Letters to the Editor* or other editorial matters should be sent to the same address listed above to the attention of *CROSSTALK Editor* or send directly to the senior editor via the E-mail address also listed above.

Article Submissions: We welcome articles of interest to the defense software community. Articles must be approved by the *CROSSTALK* editorial board prior to publication. Please follow the *Guidelines for CROSSTALK Authors*, available upon request. We do not pay for submissions. Articles published in *CROSSTALK* remain the property of the authors and may be submitted to other publications.

Reprints and Permissions: Requests for reprints must be requested from the author or the copyright holder. Please coordinate your request with *CROSSTALK*.

Trademarks and Endorsements: All product names referenced in this issue are trademarks of their companies. The mention of a product or business in *CROSSTALK* does not constitute an endorsement by the Software Technology Support Center (STSC), the Department of Defense, or any other government agency. The opinions expressed represent the viewpoints of the authors and are not necessarily those of the Department of Defense.

Coming Events: We often list conferences, seminars, symposiums, etc., that are of interest to our readers. There is no fee for this service, but we must receive the information at least 90 days before registration. Send an announcement to the *CROSSTALK* Editorial Department.

STSC On-Line Services: STSC On-Line Services can be reached on the Internet. World Wide Web access is at http://www.stsc.hill.af.mil. The STSC maintains a Gopher server at gopher://gopher.stsc.hill.af.mil. Its ftp site may be reached at ftp://ftp.stsc.hill.af.mil. The Lynx browser or gopher server can also be reached using telnet at bbs.stsc.hill.af.mil or by modem at 801-774-6509 or DSN 775-3602. Call 801-777-7989 or DSN 777-7989 for assistance, or E-mail to portr@software.hill.af.mil.

Publications Available: The STSC provides various publications at no charge to the defense software community. Fill out the Request for STSC Services card in the center of this issue and mail or fax it to us. If the card is missing, call Customer Service at the numbers shown above, and we will send you a form or take your request by phone. The STSC sometimes has extra paper copies of back issues of *CROSSTALK* free of charge. If you would like a copy of the printed edition of this or another issue of *CROSSTALK*, or would like to subscribe, please contact the customer service address listed above.

The ***Software Technology Support Center*** was established at Ogden Air Logistics Center (AFMC) by Headquarters U.S. Air Force to help Air Force software organizations identify, evaluate, and adopt technologies that will improve the quality of their software products, their efficiency in producing them, and their ability to accurately predict the cost and schedule of their delivery. *CROSSTALK* is assembled, printed, and distributed by the Defense Automated Printing Service, Hill AFB, UT 84056. *CROSSTALK* is distributed without charge to individuals actively involved in the defense software development process.

Awaiting the Big Delivery

You shouldn't wait too long to fill your children's lives with disappointment. If you shelter them too much, they'll never know how to handle life's hard turns, which is why when my daughter was born just a couple days ago, she wasn't even completely out of the womb when I decided to share what I'm sure everyone considers the most painful national disappointment since Lloyd Bentsen was replaced as secretary of the treasury: "I'm sorry, sweetie, but the CMMI initiative will delay CMM, Version 2.0 for months and may even absorb it entirely."

She cried at the news, but she's doing better now. Her nearly two-year-old brother handled the news fairly well, too, considering the impact capability maturity models (CMMs) have had on his life. For months, our bedtime reading has consisted of the latest ISO standard or someone's new CMM (Me: "Honey, he was out cold in six seconds flat—a new record!" My Wife: "No it isn't. He's faking it to make you go away."). Yet, somehow, he handled the news of this earth-shattering delay with remarkable restraint: "Oh." (yawn) [clunk!]

You'll see essentially the same reaction from software developers. But I know that somewhere, someone is miffed that they have to wait months to receive *even more* mandated best practices they really should follow, but do not. In the meantime, these people will have to wait in line at the local library in the wing set aside for all the good and not-so-good CMMs out there, including the T-CMM (Testing CMM), the P-CMM (People CMM), the VMR-CMM (Vending Machine Renewal CMM), the CMRDEDBDMNMs-CMM (See, 'Em Are De Eedy Beedy M&Ms CMM. Read it out loud. Har!), and The CMM of the Living Dead (formerly the Project Management CMM). And I'm assuming that to rate the quality and utility of all these CMMs, someone must have already developed a CMM-CMM (the Capability Maturity Model Capability Maturity Model).

In light of all the available CMMs, I think the Department of Defense has the right idea in wanting the Software Engineering Institute to combine a couple of its own CMMs into a CMM Integration (CMMI). However, this has delayed the release of the much-anticipated CMM, Version 2. The funny thing is, even among big CMM fans, I'm having a hard time finding anyone disappointed about this. The reason is simple: The Seventh Grade Pre-Algebra Principle.

Think back to your days in pre-algebra, a class of students ranging from the pre-pubescent equivalent of CMM Level 5—algebra whizzes like Joe S., the confident center of attention with his "dirty joke of the day"—to me at CMM Level 1—as relaxed and confident as a 98-lb. weakling at a sand-kicking contest, and who at year's end couldn't determine the area of a triangle any better than I understood Joe's jokes.

Now, imagine your teacher tells you the scheduled midterm exam has been postponed indefinitely. What are your emotions? Disgust? Disappointment? If so, I suppose that you were also one of the kids who left your corrected test in the corner of your desk so everyone could see the score. Maybe assessments are necessary, but if I'd heard there was going to be a delay in the next round of "tests," I'd feel like the class bully had just told me he'd decided not to beat me up after all, and that he was looking for someone to get me out of my locker.

So lack of disappointment about the CMM delay is understandable. Besides, a maturity level doesn't tell the whole story. For example, sure, Mr. Level 5 Joe S. showed up at our high school 10-year reunion as a high-ranking executive for one of the world's largest computer corporations, but what about Level 2 Sean M.? He was a success, too. In a two-page *Sports Illustrated* lead-in photo he was shown committing what must have been a second-degree felony in a story about flagrant fouls in NCAA basketball. (He also once stuck gum in my hair, although I've since reluctantly forgiven the unredeemable brute. And no, I didn't fit in my locker.) And little old me, CMM Level 1—not to toot my horn, but in college I was editor of one of the first 100 newspapers to pick up "Dilbert" in syndication.

So don't tell me there's a correlation between maturity and success. I have other examples, but I'd rather have you read them in my new Self-Rationalization CMM. Check it out in the CMM section of your local library. – Lorin May

Got an idea for BACKTALK? Send an E-mail to backtalk@stsc1.hill.af.mil